

# 26

## Heterostructure Device Simulation

---

*This chapter describes carrier transport boundary conditions for heterointerfaces.*

---

### Thermionic Emission Current

Conventional transport equations cease to be valid at a heterojunction interface, and currents and energy fluxes at the abrupt interface between two materials are better defined by the interface condition at the heterojunction. In defining thermionic current and thermionic energy flux, Sentaurus Device follows the literature [1].

---

### Using Thermionic Emission Current

To activate the thermionic current model for electrons at a region-interface (material-interface) heterojunction, the keyword `eThermionic` must be specified in the appropriate region-interface (material-interface) `Physics` section. For example:

```
Physics(MaterialInterface="GaAs/AlGaAs") {  
    eThermionic  
}
```

Similarly, to activate thermionic current for holes, the keyword `hThermionic` must be specified. The keyword `Thermionic` activates the thermionic emission model for both electrons and holes. If any of these keywords is specified in the `Physics` section for a region `Region.0`, where `Region.0` is a semiconductor, the appropriate model will be applied to each `Region.0`-semiconductor interface.

For small particle and energy fluxes across the interface, the condition of continuous quasi-Fermi level and carrier temperature is sometimes used. This option is activated by the keyword `Heterointerface` in the appropriate `Physics` section. In realistic transistors, such an approach might lead to unsatisfactory results [2].

## Chapter 26: Heterostructure Device Simulation

### Thermionic Emission Current

You can change the coefficients of the thermionic emission model in the ThermionicEmission parameter set in an interface-specific section in the parameter file:

```
RegionInterface = "regionA/regionB" {
    ThermionicEmission {
        A = 2, 2      # [1]
        B = 4, 4      # [1]
        C = 1, 1      # [1]
    }
}
```

## Thermionic Emission Model

Assume that at the heterointerface between material 1 and material 2, the conduction edge jump is positive, that is  $\Delta E_C > 0$ , where  $\Delta E_C = E_{C,2} - E_{C,1}$  (that is,  $\chi_1 > \chi_2$ ). If  $J_{n,2}$  and  $S_{n,2}$  are the electron current density and the electron energy flux density entering material 2, and  $J_{n,1}$  and  $S_{n,1}$  are the electron current density and the electron energy flux density leaving material 1, then the interface condition can be written as:

$$J_{n,2} = J_{n,1} \quad (884)$$

$$J_{n,2} = a_n q \left[ v_{n,2} n_2 - \frac{m_{n,2}}{m_{n,1}} v_{n,1} n_1 \exp \left( -\frac{\Delta E_C}{kT_{n,1}} \right) \right] \quad (885)$$

$$S_{n,2} = S_{n,1} + \frac{c_n}{q} J_{n,2} \Delta E_C \quad (886)$$

$$S_{n,2} = -b_n \left[ v_{n,2} n_2 kT_{n,2} - \frac{m_{n,2}}{m_{n,1}} v_{n,1} n_1 kT_{n,1} \exp \left( -\frac{\Delta E_C}{kT_{n,1}} \right) \right] \quad (887)$$

where the ‘emission velocities’ are defined as:

$$v_{n,i} = \sqrt{\frac{kT_{n,i}}{2\pi m_{n,i}}} \quad (888)$$

and by default, the coefficients in the above equations are  $a_n = 2$ ,  $b_n = 4$ , and  $c_n = 1$ , which corresponds to the literature [1]. Similar equations for the hole thermionic current and hole thermionic energy flux are presented below:

$$J_{p,2} = J_{p,1} \quad (889)$$

$$J_{p,2} = -a_p q \left[ v_{p,2} p_2 - \frac{m_{p,2}}{m_{p,1}} v_{p,1} p_1 \exp \left( \frac{\Delta E_V}{kT_{p,1}} \right) \right] \quad (890)$$

$$S_{p,2} = S_{p,1} + \frac{c_p}{q} J_{p,2} \Delta E_V \quad (891)$$

$$S_{p,2} = -b_p \left[ v_{p,2} p_2 kT_{p,2} - \frac{m_{p,2}}{m_{p,1}} v_{p,1} p_1 kT_{p,1} \exp \left( \frac{\Delta E_V}{kT_{p,1}} \right) \right] \quad (892)$$

## Chapter 26: Heterostructure Device Simulation

### Thermionic Emission Current

$$v_{p,i} = \sqrt{\frac{kT_{p,i}}{2\pi m_{p,i}}} \quad (893)$$

An equivalent set of equations are used if Fermi carrier statistics is selected.

## Thermionic Emission Model With Fermi Statistics

With Fermi statistics, the electron current density and the electron energy flux density are:

$$J_{n,2} = a_n q v_{n,2} N_{C,2} \zeta_{n,2} - \frac{m_{n,2}}{m_{n,1}} v_{n,1} N_{C,1} \zeta_{n,1} \quad (894)$$

$$S_{n,2} = -b_n v_{n,2} k T_{n,2} N_{C,2} \zeta_{n,2} - \frac{m_{n,2}}{m_{n,1}} v_{n,1} k T_{n,1} N_{C,1} \zeta_{n,1} \quad (895)$$

$$\zeta_{n,2} = \ln[1 + \exp(-\eta_{n,2})] + \eta_{n,2} \quad (896)$$

$$\zeta_{n,1} = \ln[1 + \exp(-\eta_{n,1}')] + \eta_{n,1}' \quad (897)$$

$$\eta_{n,1}' = \eta_{n,1} - \frac{\Delta E_C}{k T_{n,1}} \quad (898)$$

Here,  $\eta_n$  is given in [Equation 49](#) and  $N_C$  is given in [Equation 177](#). Similar equations apply to holes:

$$J_{p,2} = -a_p q v_{p,2} N_{V,2} \zeta_{p,2} - \frac{m_{p,2}}{m_{p,1}} v_{p,1} N_{V,1} \zeta_{p,1} \quad (899)$$

$$S_{p,2} = -b_p v_{p,2} k T_{p,2} N_{V,2} \zeta_{p,2} - \frac{m_{p,2}}{m_{p,1}} v_{p,1} k T_{p,1} N_{V,1} \zeta_{p,1} \quad (900)$$

$$\zeta_{p,2} = \ln[1 + \exp(-\eta_{p,2})] + \eta_{p,2} \quad (901)$$

$$\zeta_{p,1} = \ln[1 + \exp(-\eta_{p,1}')] + \eta_{p,1}' \quad (902)$$

$$\eta_{p,1}' = \eta_{p,1} + \frac{\Delta E_V}{k T_{p,1}} \quad (903)$$

Here,  $\eta_p$  is given in [Equation 50](#) and  $N_V$  is given in [Equation 181](#).

If Fermi statistics is used, this model can be activated by specifying `Formula=1` in the `ThermionicEmission` section of the parameter file:

```
ThermionicEmission { Formula=1 }
```

By default `Formula=0`, it activates the old model, where the Boltzmann-like thermionic emission equations similar to [Equation 884–Equation 893](#) are used. This can lead to incorrect results in high carrier density. In this case, a warning message will be given.

**Note:**

Always specify `Formula=1` when Fermi statistics is used. If Fermi statistics is not used, [Equation 884–Equation 893](#) will be activated regardless of the `Formula` statement in the parameter file.

---

## Gaussian Transport Across Organic Heterointerfaces

A thermionic-like current boundary condition has been introduced to correctly account for carrier transport across organic heterointerfaces. An organic heterointerface is defined in this context as an heterointerface with the Gaussian density-of-states (DOS) model (see [Gaussian Density-of-States for Organic Semiconductors on page 322](#)) activated in both regions that are adjacent to the heterointerface.

---

### Using Gaussian Transport at Organic Heterointerfaces

The model is activated by switching to the Gaussian DOS model in both regions of the heterointerface that are meant to be organic:

```
Physics(Region="OrganicRegion_1") {
    EffectiveMass(GaussianDOS)
}
Physics(Region="OrganicRegion_2") {
    EffectiveMass(GaussianDOS)
}
```

and then specifying the keyword `Organic_Gaussian` as an option for the `Thermionic` model in the `Physics` section of the organic heterointerface:

```
Physics(RegionInterface="OrganicRegion_1/OrganicRegion_2") {
    Thermionic(Organic_Gaussian)
}
```

This syntax also automatically switches on the double points at the organic heterointerface.

You can adjust the parameters  $v_{n,\text{org}}$  and  $v_{p,\text{org}}$  in [Equation 905](#) and [Equation 907](#) in the `ThermionicEmission` section of the parameter file (their default values are  $1 \times 10^6$  cm/s):

```
RegionInterface="OrganicRegion_1/OrganicRegion_2" {
    ThermionicEmission {
        vel_org = 1e7, 1e7    # [cm/s]
    }
}
```

## Gaussian Transport at Organic Heterointerface Model

Assuming a positive conduction edge jump and a negative valence edge jump from material 1 to material 2, the boundary conditions at the organic heterointerface are given by:

$$J_{n,2} = J_{n,1} \quad (904)$$

$$J_{n,2} = v_{n,\text{org}} q n_2 - n_1 \exp -\frac{\Delta E_C}{kT_{n,1}} \quad (905)$$

$$J_{p,2} = J_{p,1} \quad (906)$$

$$J_{p,2} = v_{p,\text{org}} q p_2 - p_1 \exp -\frac{\Delta E_V}{kT_{p,1}} \quad (907)$$

where:

- $J_{n,2}$  and  $J_{n,1}$  are the electron current densities entering material 2 and leaving material 1, respectively.
- $J_{p,2}$  and  $J_{p,1}$  are the hole current densities leaving material 2 and entering material 1, respectively.
- $\Delta E_C = d_{C,2} - d_{C,1}$  where  $d_{C,2}$  and  $d_{C,1}$  are the distances of the Gaussian distribution peaks to the conduction band edges.
- $\Delta E_V = d_{V,2} - d_{V,1}$  where  $d_{V,2}$  and  $d_{V,1}$  are the distances of the Gaussian distribution peaks to the valence band edges.

## References

- [1] D. Schroeder, *Modelling of Interface Carrier Transport for Device Simulation*, Wien: Springer, 1994.
- [2] K. Horio and H. Yanai, "Numerical Modeling of Heterojunctions Including the Thermionic Emission Mechanism at the Heterojunction Interface," *IEEE Transactions on Electron Devices*, vol. 37, no. 4, pp. 1093–1098, 1990.

# 27

## Energy-Dependent Parameters

---

*This chapter describes extensions for the temperature-dependent models.*

---

### Overview

Sentaurus Device provides the possibility to specify some parameters as a ratio of two irrational polynomials. The general form of such ratio is written as:

$$G(w, s) = f \frac{((a_i w^{p_i}) + d_n s)^{g_n}}{((a_j w^{p_j}) + d_d s)^{g_d}} \quad (908)$$

where:

- Subscripts  $n$  and  $d$  correspond to the numerator and denominator, respectively.
- $f$  is a factor,  $w$  is a primary variable, and  $s$  is an additional variable.

You can use [Equation 908](#) with different coefficients for different intervals  $k$  defined by the segment  $[w_{k-1}^{\max}, w_k^{\max}]$ . By default, it is assumed that only one interval  $k = 0$  with the boundaries  $[0, \infty]$  exists, and function  $G$  is constant, that is,  $a_0 = 0$ ,  $a_i = 0$ ,  $p = d = 0$ ,  $g = 1$ . Factor  $f$  is defined accordingly for each model.

A simplified syntax is introduced to define the piecewise linear function  $G$ . The boundaries of the intervals and the value of factor must be specified, which means the value of  $G$  is at the right side of the interval. All other coefficients should not be specified to use this possibility. As there are some peculiarities in parameter specification and model activation, the specific models for which the approximation by [Equation 908](#) is supported are described here separately.

## Chapter 27: Energy-Dependent Parameters

### Energy-Dependent Energy Relaxation Time

## Energy-Dependent Energy Relaxation Time

For the specification of the energy relaxation time, the following modification of [Equation 908](#) is used:

$$\tau(w) = \tau_w^0 \frac{((a_i w^{p_i}))^{g_n}}{((a_j w^{p_j}))^{g_d}} \quad (909)$$

where  $w = 1.5kT_n/q$  for electrons and  $w = kT_p/q$  for holes.

The factor  $f$  in [Equation 908](#) is defined by  $\tau_w^0$ , which can be specified in the parameter file by the values `tau_w_ele` and `tau_w_hol`.

To activate the specification of the energy-dependent energy relaxation time, the parameter `Formula(tau_w_ele)` (or `Formula(tau_w_hol)` for holes) must be set to 2.

The following example shows the energy relaxation time section of the parameter file and provides a short description of the syntax:

```
EnergyRelaxationTime
{ * Energy relaxation times in picoseconds
  tau_w_ele = 0.3    # [ps]
  tau_w_hol = 0.25   # [ps]
* Below is the example of energy relaxation time approximation
* by the ratio of two irrational polynomials.
* If Wmax(interval-1) < Wc < Wmax(interval), then:
* tau_w = (tau_w)*(Numerator^Gn)/(Denominator^Gd),
* where (Numerator or Denominator)=SIGMA[A(i)(Wc^P(i))],
* Wc=1.5(k*Tcar)/q (in eV).
* By default: Wmin(0)=Wmax(-1)=0; Wmax(0)=infinity.
* The option can be activated by specifying appropriate Formula
* equals 2
*     Formula(tau_w_ele) = 2
*     Formula(tau_w_hol) = 2
*     Wmax(interval)_ele =
*     tau_w_ele(interval) =
*     Numerator(interval)_ele{
*       A(0) =
*       P(0) =
*       A(1) =
*       P(1) =
*       D   =
*       G   =
*     }
*     Denominator(interval)_ele{
*       A(0) =
*       P(0) =
*       D   =
*       G   =
```

## Chapter 27: Energy-Dependent Parameters

### Energy-Dependent Energy Relaxation Time

```
*      }
*      Wmax(interval)_hol =
*      tau_w_hol(interval) =
tau_w_ele = 0.3    # [ps]
tau_w_hol = 0.25  # [ps]

Formula(tau_w_ele) = 2
Numerator(0)_ele{
    A(0) = 0.048200
    P(0) = 0.00
    A(1) = 1.00
    P(1) = 3.500
    A(2) = 0.0500
    P(2) = 2.500
    A(3) = 0.0018100
    P(3) = 1.00
}
Denominator(0)_ele{
    A(0) = 0.048200
    P(0) = 0.00
    A(1) = 1.00
    P(1) = 3.500
    A(2) = 0.100
    P(2) = 2.500
}
```

The following example shows a simplified syntax for piecewise linear specification of energy relaxation time:

```
EnergyRelaxationTime:
{ * Energy relaxation times in picoseconds
    Formula(tau_w_ele) = 2
    tau_w_ele = 0.3          # [ps]

    Wmax(0)_ele = 0.5        # [eV]
    tau_w_ele(1) = 0.46       # [ps]

    Wmax(1)_ele = 1.0         # [eV]
    tau_w_ele(2) = 0.4         # [ps]

    Wmax(2)_ele = 2.0         # [eV]
    tau_w_ele(3) = 0.2         # [ps]

    tau_w_hol = 0.25          # [ps]
}
```

---

## Spline Interpolation

Sentaurus Device also allows spline approximation of energy relaxation time over energy. In this case, the parameter `Formula(tau_w_ele)` for electron energy relaxation time (and

## Chapter 27: Energy-Dependent Parameters

### Energy-Dependent Mobility

similarly, parameter `Formula(tau_w_hol)` for hole energy relaxation time) must be equal to 3.

Inside the braces following the keyword `Spline(tau_w_ele)` (or `Spline(tau_w_hol)`), an energy [eV] and tau [ps] value pair must be specified in each line. For the values outside of the specified intervals, energy relaxation time is treated as a constant and equal to the closest boundary value.

The following example shows a spline approximation specification for energy-dependent energy relaxation time for electrons:

```
EnergyRelaxationTime {
    Formula(tau_w_ele) = 3
    Spline(tau_w_ele) {
        0.      0.3   # [eV] [ps]
        0.5     0.46  # [eV] [ps]
        1.      0.4   # [eV] [ps]
        2.      0.2   # [eV] [ps]
    }
}
```

#### Note:

Energy relaxation times can be either energy-dependent or mole fraction-dependent (see [Abrupt and Graded Heterojunctions on page 59](#)), but not both.

---

## Energy-Dependent Mobility

In addition to existing energy-dependent mobility models (such as Caughey–Thomas, where the effective field is computed inside Sentaurus Device as a function of the carrier temperature), a more complex, user-supplied mobility model can be defined. For such a specification of energy-dependent mobility, a modification to [Equation 908](#) is used:

$$\mu(\bar{w}, N_{\text{tot}}) = \mu_{\text{low}} \frac{((a_i \bar{w}^{p_i}) + d_n N_{\text{tot}})^{g_n}}{((a_j \bar{w}^{p_j}) + d_d N_{\text{tot}})^{g_d}} \quad (910)$$

where  $\bar{w} = T_n/T$  for electrons or  $\bar{w} = T_p/T$  for holes, and  $\mu_{\text{low}}$  is the low field mobility.

To activate the model, you must specify `CarrierTemperatureDrivePolynomial`, the

driving force keyword, as a parameter of the high-field saturation mobility model.

Parameters of the polynomials must be defined in the `HydroHighFieldMobility` parameter set.

## Chapter 27: Energy-Dependent Parameters

### Energy-Dependent Mobility

This example shows the output of the `HydroHighFieldMobility` section and the specification of coefficients:

```
HydroHighFieldDependence:  
{ * Parameter specifications for the high field degradation in  
* some hydrodynamic models.  
* B) Approximation by the ratio of two irrational polynomials  
* (driving force 'CarrierTempDrivePolynomial'):  
* If Wmax(interval-1) < w < Wmax(interval), then:  
* mu_hf = mu*factor*(Numerator^Gn)/(Denominator^Gd),  
* where (Numerator or Denominator)={SIGMA[A(i)(w^P(i))] + D*Ni},  
* w=Tc/Tl; Ni(cm^-3) is total doping.  
* By default: Wmin(0)=Wmax(-1)=0; Wmax(0)=infinity.  
  
* Wmax(interval)_ele =  
* F(interval)_ele =  
* Numerator(interval)_ele{  
*   A(0) =  
*   P(0) =  
*   A(1) =  
*   P(1) =  
*   D =  
*   G =  
* }  
* Denominator(interval)_ele{  
*   A(0) =  
*   P(0) =  
*   D =  
*   G =  
* }  
* F(interval)_hol =  
* Wmax(interval)_hol =  
* Denominator(0)_ele  
{  
  A(0) = 0.3  
  P(0) = 0.0  
  A(1) = 1.0  
  P(1) = 2.  
  A(2) = 0.001  
  P(2) = 2.500  
  D = 3.00e-16  
  G = 0.2500  
}  
}
```

---

## Spline Interpolation

Instead of the rational polynomial in [Equation 910](#), you can use a spline interpolation. In this case, the option `CarrierTempDriveSpline` must be used for the high-field mobility model in the command file.

## Chapter 27: Energy-Dependent Parameters

### Energy-Dependent Peltier Coefficient

For example:

```
Physics {
    Mobility (
        HighFieldSaturation (CarrierTempDriveSpline)
    )
}
```

The energy-dependent mobility is computed as:

$$\mu(\bar{w}) = \mu_{\text{low}} \cdot \text{spline}(\bar{w}) \quad (911)$$

where the function  $\text{spline}(\bar{w})$  is defined by a sequence of value pairs in the parameter file:

```
HydroHighFieldDependence {
    Spline (electron) {
        0   1
        1   1
        2   2.5
        4   4
        10  5
    }

    Spline (hole) {
        0   1
        1   1
        2   0.75
        4   0.5
        10  0.2
    }
}
```

The given data points are interpolated by a cubic spline. Zero derivatives are imposed as boundary conditions at the end points. The spline function remains constant beyond the end points.

---

## Energy-Dependent Peltier Coefficient

Sentaurus Device allows for the following modification of the expression of the energy flux equation:

$$\vec{S}_n = -\frac{5r_n}{2} \frac{kT_n}{q} \vec{J}_n + f_n^{\text{hf}} \hat{\kappa}_n \frac{\partial(w_n \Pi_n)}{\partial w_n} \nabla T_n \quad (912)$$

The standard expression corresponds to  $\Pi_n = 1$ . If  $\Pi_n = 1 + P(\bar{w})$ , then:

$$\frac{\partial(w_n \Pi_n)}{\partial w_n} = 1 + \bar{w} \frac{\partial P(\bar{w})}{\partial \bar{w}} C \quad (913)$$

## Chapter 27: Energy-Dependent Parameters

### Energy-Dependent Peltier Coefficient

Sentaurus Device allows you to specify the function  $Q$ :

$$Q(\bar{w}) = \bar{w} \frac{\partial P(\bar{w})}{\partial \bar{w}} \quad (914)$$

For the specification of  $Q$ , the following modification of [Equation 908](#) is used:

$$Q(\bar{w}) = f \frac{\left( \left( -a_i \bar{w}^{p_i} \right) \right)^{g_n}}{\left( \left( -a_j \bar{w}^{p_j} \right) \right)^{g_d}} \quad (915)$$

Coefficients must be specified in the `HeatFlux` parameter set, and the dependence can be activated by specifying a nonzero factor  $f$ .

For  $\Pi_n = 1 + 1/\sqrt{1 + \bar{w}^2}$ , the result is  $Q = 1/(\bar{w}^2 + 1)^{1.5}$ .

This is an example of the parameter file section for such a function  $Q_n$ :

```

HeatFlux
{ * Heat flux factor (0 <= hf <= 1)
  hf_n = 1      # [1]
  hf_p = 1      # [1]
* Coefficients can be defined also as:
*   hf_new = hf*(1.+Delta(w))
* where Delta(w) is the ratio of two irrational polynomials.
* If Wmax(interval-1) < Wc < Wmax(interval), then:
* Delta(w) = factor*(Numerator^Gn)/(Denominator^Gd),
* where (Numerator or Denominator)=SIGMA[A(i)(w^P(i))], w=Tc/Tl
* By default: Wmin(0)=Wmax(-1)=0; Wmax(0)=infinity.
* Option can be activated by specifying nonzero 'factor'.
*   Wmax(interval)_ele =
*   F(interval)_ele = 1
*   Numerator(interval)_ele{
*     A(0)  =
*     P(0)  =
*     A(1)  =
*     P(1)  =
*     G     =
*   }
*   Denominator(interval)_ele{
*     A(0)  =
*     P(0)  =
*     G     =
*   }
*   Wmax(interval)_hol =
*   F(interval)_hol = 1
*   f(0)_ele = 1
*   Denominator(0)_ele{
*     A(0)  = 1.
*     P(0)  = 0.
*     A(1)  = 1.

```

## Chapter 27: Energy-Dependent Parameters

### Energy-Dependent Peltier Coefficient

```
P(1) = 2.  
G     = 1.5  
}
```

---

## Spline Interpolation

Instead of the rational polynomial in [Equation 915](#), a spline interpolation can be used as well. The function  $Q(w)$  is defined in the parameter file by a sequence of value pairs:

```
HeatFlux {  
    hf_n = 1  
    hf_p = 1  
  
    Spline (electron) {  
        0   1  
        1   1  
        2   2.5  
        4   4  
       10  5  
    }  
  
    Spline (hole) {  
        0   1  
        1   1  
        2   0.75  
        4   0.5  
       10  0.2  
    }  
}
```

The given data points are interpolated by a cubic spline. Zero derivatives are imposed as boundary conditions at the end points. The spline function remains constant beyond the end points.

# 28

## Anisotropic Properties

---

This chapter discusses the anisotropic properties of semiconductor devices.

### Anisotropic Properties of Semiconductor Devices

In general, all equations for semiconductor devices can be written in the following form:

$$-\nabla \cdot \vec{J} = R \quad (916)$$

Here, vector  $\vec{J} = \hat{\mu} \vec{g}$  and the tensor coefficient  $\hat{\mu} = \hat{\mu} \cdot \hat{A}$ , where  $\hat{\mu}$  is a scalar function, tensor  $\hat{A}$  is a  $3 \times 3$  (or  $2 \times 2$ , in two dimensions) symmetric matrix, and vector  $\vec{g}$  is a first-order differential expression.

In the isotropic case, tensor  $\hat{A}$  is the unit matrix. In the common case, tensor  $\hat{A}$  depends on the solution. Among the reasons why tensor  $\hat{A}$  is a full matrix are the anisotropic properties of semiconductors (such as SiC) and the influence of mechanical stress, which affects anisotropic mobility (see [Chapter 31 on page 935](#)).

Table 150 Anisotropic models for the tensor coefficient  $\hat{\mu}$  that Sentaurus Device supports

Tensor coefficient	Equation or model
Mobility, $\hat{\mu}$	Continuity equation for electrons and holes
Electrical permittivity, $\hat{\epsilon}$	Poisson equation
Thermal conductivity, $\hat{\kappa}$	Thermodynamics equation
Quantum potential parameter, $\hat{\alpha}$	Density gradient model

## Chapter 28: Anisotropic Properties

### Anisotropic Properties of Semiconductor Devices

Sentaurus Device provides different anisotropic approximations for discretization for the Poisson, continuity, and thermodynamic equations, and for the density gradient model (see [Anisotropic Approximations on page 884](#)).

#### Note:

Anisotropic properties might have not only the tensor coefficient  $\hat{\mu}$ , but also some generation–recombination models (see [Anisotropic Avalanche Generation on page 899](#)).

---

## Anisotropic Approximations

Due to the complexity of anisotropic simulations, Sentaurus Device offers different approximations:

- The `TensorGridAniso` approximation (default) is the most robust, but it has some accuracy issues for nonaxis-aligned meshes or when the anisotropy orientation does not coincide with the mesh orientation.
- The `AnisoSG` approximation gives the most accurate results and is independent of the mesh orientation. Convergence, however, might be worse than for `TensorGridAniso`, for example.
- The `StressSG` approximation gives the most accurate results and is independent of the mesh orientation. Convergence, however, might be worse than for `TensorGridAniso`.
- The accuracy of the `AverageAniso` approximation depends on the Delaunay properties of the virtual mesh obtained after anisotropic transformation. If this mesh is a Delaunay mesh, the results are relatively accurate. Otherwise, the accuracy degrades.

## TensorGridAniso

This is the default approximation and it is correct only for tensor grids or grids close to tensor ones.

The anisotropic effects are modeled using a tensor-grid approximation. The eigenvalues and eigenvectors of tensor  $A$  are used as multiplication factors for the projections of vector  $\vec{g}$  on mesh edges.

The following options in the `Math` section activate this approximation:

```
Math {
    TensorGridAniso          # only for stress mobility
    TensorGridAniso(Piezo)    # stress problems, same as above
    TensorGridAniso(Aniso)    # anisotropic models, see this chapter
    TensorGridAniso(Aniso Piezo) # anisotropic models and stress
                                # mobility
}
```

## AnisoSG

The anisotropic Scharfetter–Gummel (`AnisoSG`) approximation is available for the Poisson, continuity, and thermodynamics equations. If the anisotropic direction is inconsistent with the mesh, the `AverageAniso` and `TensorGridAniso` approximations might not be accurate enough (*mesh orientation effect*). The `AnisoSG` approximation is not dependent on the mesh orientation.

Traditionally, the Scharfetter–Gummel approximation is used for the continuity equation (see [Chapter 38 on page 1177](#)), where the argument of the Bernoulli function  $x^* = (\vec{E}^*, \vec{l})$  is a projection of the effective field  $\vec{E}^*$  on edge  $\vec{l}$ . If, for the isotropic case,  $x^*$  depends on values at edge nodes only, for an anisotropic problem, it is necessary to compute vector  $\vec{E}^*$  elementwise. This concept allows for the generalization of the Scharfetter–Gummel approximation to the anisotropic case.

To activate the `AnisoSG` approximation, specify the keyword `AnisoSG` in the global `Math` section:

```
Math { AnisoSG }
```

The `AnisoSG` branch, if it converges, guarantees that concentrations remain positive. However, sometimes, there might be convergence problems due to the unstable computation of the argument of the Bernoulli function  $x^*$  and corresponding derivatives, when the carrier concentration is very small. To mitigate this problem, you can use the following parameters:

- `AnisoSG_MinDen` is a stabilization parameter for the computation of  $x^*$ . Its default value is  $10 \text{ cm}^{-3}$ .
- `AnisoSG_DerivativeMinDen` specifies the lower limit of carrier densities for the `AnisoSG` approximation. Corresponding derivatives are switched off if the node carrier density is below this value. Its default value is zero.

You can modify these values in the `Math` section:

```
Math {
    AnisoSG
    AnisoSG_MinDen = 1e6          # [cm-3]
    AnisoSG_DerivativeMinDen = 1e2 # [cm-3]
}
```

## StressSG

This approximation is implemented for stress problems and it affects anisotropic mobility (see [Chapter 31 on page 935](#)). To activate this approximation, specify the `StressSG` keyword in the global `Math` section:

```
Math { StressSG }
```

## AverageAniso

This approximation uses a local (vertex-wise) linear transformation, which transforms an anisotropic problem to an isotropic one. After this, Sentaurus Device uses the `AverageBoxMethod` algorithm to compute control volumes and coefficients (see [Chapter 38 on page 1177](#)). The keyword `AverageAniso` in the `Math` section activates this approximation. Due to the requirements of `AverageBoxMethod`, the `AverageAniso` approximation requires that either the input mesh consists of triangles only (tetrahedra in three dimensions) or the mesh is tensorial, with the main axes of this tensor mesh and the main axes of the anisotropy aligned to the simulation coordinate system.

---

## Crystal and Simulation Coordinate Systems

Sentaurus Device uses the simulation coordinate system (mesh geometry) and the crystal coordinate system. Anisotropic material parameters are defined in the crystal coordinate system.

The x-axis and y-axis of the simulation coordinate system are defined in the `LatticeParameters` section in the parameter file:

```
LatticeParameters {  
    X = (1, 0, 0)  
    Y = (0, 0, -1)  
}
```

The z-axis is computed as the outer vector product of the x-axis and y-axis. The simulation coordinate system is defined relative to the crystal coordinate system. If the keyword `CrystalAxis` is present, the crystal coordinate system is defined relative to the simulation coordinate system (see [Using Stress and Strain on page 937](#)).

In fact, the `x` and `y` vectors indicate the transformation that must be applied between the crystal and simulation coordinate systems to correctly compute physical quantities that are related to the given crystal orientation, such as stress, anisotropy, and carrier quantization.

In the example, the x-axis of the simulation coordinate system coincides with the x-axis of the crystal coordinate system. The y-axis of the simulation coordinate system runs along the negative z-axis of the crystal coordinate system. This is a common definition for 2D simulations of `Piezoelectric_Polarization` (see [Chapter 31 on page 935](#)).

Instead of the `LatticeParameters` section, the keywords `Piezo` and `PiezoParameters` are recognized as well. By default, Sentaurus Device uses `X=(1,0,0)`, `Y=(0,1,0)`, and `Z=(0,0,1)`.

Sentaurus Device reads the `LatticeParameters` section (`x` and `y` vectors) from a parameter file or directly from a TDR file. If both are found, parameters from the parameter file have a higher priority. If no `LatticeParameters` section is found, then the default values are used.

**Caution:**

In the 2D case, there is a difference between how Sentaurus Device interprets the `x` and `y` vectors read from a parameter file or directly from a TDR file. All the general parameter files stored in the folder `$STROOT/tcad/$STRELEASE/lib/sdevice/MaterialDB/` have no definition of the `x` and `y` vectors.

Sentaurus Process always uses the 3D crystal coordinate system and creates the same `LatticeParameters` section for 2D and 3D cases. If you define the `x` and `y` vectors in the parameter file for a 2D problem, then Sentaurus Device uses a special agreement regarding coordinate systems for 2D problems (see [Anisotropic Direction in 2D Cases on page 888](#)).

## Anisotropy Definition and Specification

Sentaurus Device supports a kind of two-fold anisotropy in which model parameters can be specified separately along a single anisotropic direction and within the plane normal to that direction. Materials with this type of anisotropy are sometimes described as having *cylindrical symmetry*. However, this should not be confused with the 2D cylindrical coordinate system that is available for modeling 3D devices with cylindrical symmetry.

$$A = Q \begin{bmatrix} e \\ e \\ e_a \end{bmatrix} Q^T \quad \text{in three dimensions} \quad (917)$$

$$A_{2:2} = Q_{2:2} \begin{bmatrix} e \\ e_a \end{bmatrix} Q_{2:2}^T \quad \text{in two dimensions}$$

where:

- $e, e_a$  are eigenvalues of  $A$ .
- $Q$  is the  $3 \times 3$  orthogonal matrix from eigenvectors of  $A$ :  $Q = \begin{bmatrix} \vec{A}_x & \vec{A}_y & \vec{A}_z \end{bmatrix}$ .
- The quantity  $A_{2:2}$  denotes the leading  $2 \times 2$  submatrix of  $A$ .

## Anisotropic Direction in 3D Cases

The vector  $\vec{A}_z$  defines the anisotropic direction in the simulation coordinate system. This direction can be defined in the `Aniso` subsection of the command file in the crystal coordinate system (default) or the simulation coordinate system. The default value is the z-axis (the y-axis in the 2D case).

For example:

```
Physics {
    Aniso( direction = (1, 1, 0) )
        # anisotropic direction in crystal coordinate system
    Aniso( direction(CrystalSystem) = (1, 1, 0) )      # same as above

    # anisotropic direction in simulation coordinate system
    Aniso(Mobility direction(SimulationSystem) = (0, 1, -1) )
}
```

A symbolic definition such as `direction=zAxis (xAxis or yAxis)` is acceptable. This vector defines the anisotropic direction for all anisotropic models except for the density gradient model (see [Anisotropic Directions for Density Gradient Model on page 903](#)).

---

## Anisotropic Direction in 2D Cases

The main rule for anisotropic direction in 2D cases is the following:

**The direction of anisotropy in 2D cases must lie in the plane of simulation.**

Let `aniso_dir` be the 3D vector of anisotropy in the simulation coordinate system.

First, Sentaurus Device computes `aniso_dir=(ax,ay,az)`. If the `az` component does not equal zero, then Sentaurus Device prints an error message.

Second, Sentaurus Device uses one of the following ways to find the anisotropic direction:

- The `Aniso` subsection with the direction definition (keyword `direction`) in the command file (higher priority) (see [Aniso Subsection With Direction Definition](#))
- The `x` and `y` vectors from a parameter file or from a TDR file (see [Aniso Subsection Without Direction Definition on page 890](#))

## Aniso Subsection With Direction Definition

In this case, the `x` and `y` vectors and `direction` vector can be 3D vectors (but the `z`-component of the `aniso_dir` vector must be zero).

The following examples show correct and incorrect definitions.

### Example 1: SimulationSystem Option

In this case, there is no dependence on the `LatticeParameters` section.

```
Aniso( direction(SimulationSystem)=(ax ay 0) )      # correct
Aniso( direction(SimulationSystem)=(ax ay az) )      # incorrect
```

### Example 2: CrystalSystem Option (Default)

There is a dependence on the `LatticeParameters` section.

```
Aniso( direction(CrystalSystem)=(cx cy cz) )
```

In this case, Sentaurus Device uses the `vector=(cx cy cz)` and the `x` and `y` vectors to compute `aniso_dir=(ax,ay,az)`. If `az=0`, then the definition is correct. Otherwise, the definition is incorrect.

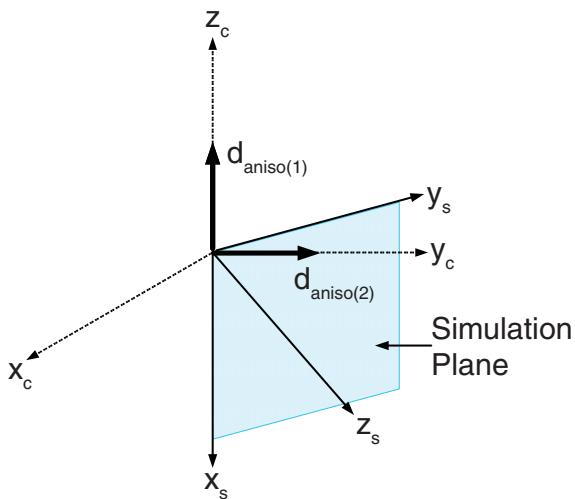
### Example 3: LatticeParameters Section

Let the `LatticeParameters` section be the following:

```
LatticeParameters {  
    X = ( 0.0000e+00, 0.0, -1.0)  
    Y = (-8.6603e-01, 0.5, 0.0)  
}
```

[Figure 54](#) shows the 2D simulation plane relative to the crystal coordinate system resulting from this specification. Two attempted anisotropic direction specifications using the `CrystalSystem` specification are also shown.

*Figure 54 Crystal coordinate system, indicated by  $x_c$ ,  $y_c$ ,  $z_c$ , and simulation coordinate system, indicated by  $x_s$ ,  $y_s$ ,  $z_s$ , as specified by the given `LatticeParameters` section. The plane of the simulation system is light blue and two specifications of the anisotropic direction are indicated.*



## Chapter 28: Anisotropic Properties

### Anisotropy Definition and Specification

To compute the anisotropic direction relative to the simulation coordinate system ( $a_x, a_y, a_z$ ) from a direction specified in the crystal coordinate system ( $c_x, c_y, c_z$ ), the following transformation is used:

$$\begin{array}{ccc} a_x & \overset{\rightarrow}{X} & c_x \\ a_y & = & \cdot \quad \overset{\rightarrow}{Y} \quad c_y \\ a_z & & \overset{\rightarrow}{Z} \quad c_z \end{array}$$

where  $\overset{\rightarrow}{X}$  and  $\overset{\rightarrow}{Y}$  are row vectors as specified in the `LatticeParameters` section, and  $\overset{\rightarrow}{Z}$  is computed automatically to form an orthogonal simulation coordinate system.

The following examples specify the anisotropic direction relative to the crystal coordinate system:

- **Case 1:** `Aniso(direction(CrystalSystem)=(0 0 1))`

As shown in [Figure 54](#), this specification causes the anisotropic direction to lie along the negative x-direction in the simulation coordinate system, that is, `aniso_dir= (-1 0 0)`. Because this direction lies within the plane of the simulation coordinate system, it is a valid specification of the anisotropic direction.

- **Case 2:** `Aniso(direction(CrystalSystem)=(0 1 0))`

As shown in [Figure 54](#), this specification causes the anisotropic direction to lie outside of the 2D simulation coordinate system, that is, `aniso_dir= (0, 0.5, 0.86603)`. Because this direction does not lie within the 2D simulation coordinate system, it is considered an invalid specification.

## Aniso Subsection Without Direction Definition

In this case, Sentaurus Device uses only the `x` and `y` vectors to compute `aniso_dir=(ax,ay,az)`. If `az=0`, then the definition is correct. Otherwise, the definition is incorrect.

Sentaurus Device interprets the `x` and `y` vectors differently in the following cases:

- Vectors are read from a parameter file (see [X and Y Vectors Read From Parameter File](#)).
- Vectors are read from a TDR file (see [X and Y Vectors Read From TDR File \(Sentaurus Process\)](#)).

## X and Y Vectors Read From Parameter File

All the general parameter files have no definition of x and y vectors. If you define the x and y vectors in the parameter file for a 2D problem, then Sentaurus Device uses a special agreement regarding coordinate systems in two dimensions:

- In the crystal coordinate system and simulation coordinate system, there are only two axes:

```
{Xc, Yc}  
{Xs, Ys}
```

- The y-axis of the crystal coordinate system is the direction of anisotropy.
- The x and y vectors must be two-dimensional vectors, that is, the third coordinate must equal zero. It means that only the next definition is correct in 2D cases:

```
LatticeParameters {  
    X = (x1, x2, 0)  
    Y = (y1, y2, 0)  
}
```

## X and Y Vectors Read From TDR File (Sentaurus Process)

Sentaurus Process always uses the 3D crystal coordinate system and creates the same LatticeParameters section for 2D and 3D cases. The z-axis of the crystal coordinate system is the direction of anisotropy.

```
LatticeParameters {  
    X = (x1, x2, x3)  
    Y = (y1, y2, y3)  
    # Z = (z1, z2, z3) is a vector product of X and Y, that is, Z=[X,Y]  
}
```

In this case, `aniso_dir=(x3, y3, z3)`. If `z3=0`, then the definition is correct. Otherwise, the definition is incorrect.

For example, using the LatticeParameters section from [Example 3: LatticeParameters Section on page 889](#):

```
aniso_dir=Z=[X,Y]=(0.5, 8.6603e-01, 0.0)
```

If `z3!=0`, then the definition is correct.

However, for example:

```
LatticeParameters {  
    X = ( 0.0348782, 0.0604109, -0.99756)  
    Y = (-0.86603, 0.5, 0.0000e+00)  
    # Z=[X,Y]= ( 0.498782, 0.863916, 0.0697565)  
}
```

In this case, `aniso_dir=(-0.99756, 0, 0.06976)`. If  $z_3 \neq 0$ , then the definition is incorrect.

---

## Aniso(direction) Versus LatticeParameters

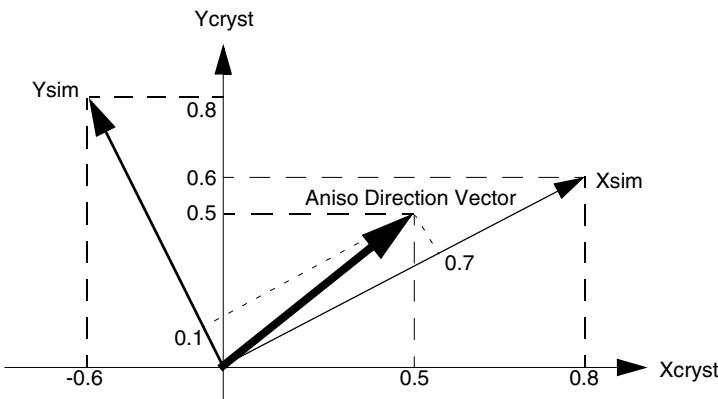
If you know the direction of anisotropy in the simulation coordinate system, then the keyword `direction` with the `SimulationSystem` option is the simplest way to define the anisotropic direction. The next examples are equivalent.

### Example 1

```
Parameter file:  
LatticeParameters {  
    X = ( 0.8, 0.6, 0 )  
    Y = (-0.6, 0.8, 0 )  
}  
  
Input command file:  
Physics {  
    Aniso( direction=(0.5, 0.5) )      # in crystal coordinate system  
}
```

### Example 2

```
Input command file:  
Physics {  
    # No dependence on LatticeParameters  
    Aniso( direction(SimulationSystem)=(0.7, 0.1) )  
}
```



See [Anisotropic Directions for Density Gradient Model on page 903](#).

## Orthogonal Matrix From Eigenvectors Q

Let  $A = (x, y, z)$  be a vector row and let  $\vec{A} = (x, y, z)'$  be a vector column.

If the anisotropic direction has the default value (z-axis in three dimensions or y-axis in two dimensions), then the matrix  $Q$  is:

$$Q = R^T, R = \begin{bmatrix} \vec{x} & \vec{y} & \vec{z} \\ \| \vec{x} \| & \| \vec{y} \| & \| \vec{z} \| \end{bmatrix} \quad (918)$$

where the vector rows  $x$ ,  $y$ , and  $z$  are defined in the `LatticeParameters` section.

If the anisotropic direction is a vector  $\vec{A}_d = (x, y, z)'$  in the crystal coordinate system, then the matrix  $Q$  is:

$$Q = \begin{bmatrix} \vec{A}_x & \vec{A}_y & \vec{A}_z \end{bmatrix}, \vec{A}_z = R^T \cdot \vec{A}_d / \| \vec{A}_d \| \quad (919)$$

If the anisotropic direction is a vector  $\vec{A}_d = (x, y, z)'$  in the simulation coordinate system, then the matrix  $Q$  is:

$$Q = \begin{bmatrix} \vec{A}_x & \vec{A}_y & \vec{A}_z \end{bmatrix}, \vec{A}_z = \vec{A}_d / \| \vec{A}_d \| \quad (920)$$

Sentaurus Device computes the vectors  $\vec{A}_x, \vec{A}_y$  to ensure that the matrix  $Q$  is orthogonal.

## Anisotropic Mobility

In some semiconductors, such as silicon carbide, electrons and holes can exhibit different mobilities along different crystallographic axes.

## Anisotropy Factor

In a 3D simulation, Sentaurus Device assumes that electrons or holes exhibit a mobility  $\mu$  along the x-axis and y-axis, and an anisotropic mobility  $\mu_{\text{aniso}}$  along the z-axis. In a 2D simulation, the regular mobility  $\mu$  is observed along the x-axis, and  $\mu_{\text{aniso}}$  is observed along the y-axis. The anisotropy factor  $r$  is defined as the ratio:

$$r = \frac{\mu}{\mu_{\text{aniso}}} \quad (921)$$

## Chapter 28: Anisotropic Properties

### Anisotropic Mobility

---

## Current Densities

In the isotropic case, current densities can be expressed by:

$$\vec{J}_n = \mu_n \vec{g}_n \quad (922)$$

$$\vec{J}_p = \mu_p \vec{g}_p \quad (923)$$

where  $\vec{g}_n$  and  $\vec{g}_p$  are the *currents without mobilities*.

In the drift-diffusion model, you have:

$$\vec{g}_n = -nq \nabla \Phi_n \quad (924)$$

$$\vec{g}_p = -pq \nabla \Phi_p \quad (925)$$

as can be seen from [Equation 60](#) and [Equation 61](#). For the thermodynamic model, [Equation 62](#) and [Equation 63](#) imply that:

$$\vec{g}_n = -nq(\nabla \Phi_n + P_n \nabla T) \quad (926)$$

$$\vec{g}_p = -pq(\nabla \Phi_p + P_p \nabla T) \quad (927)$$

In the hydrodynamic case, you have:

$$\vec{g}_n = n \nabla E_C + kT_n \nabla n + f_n^{\text{td}} kn \nabla T_n - 1.5nkT_n \nabla \ln m_n \quad (928)$$

$$\vec{g}_p = p \nabla E_V - kT_p \nabla p - f_p^{\text{td}} kp \nabla T_p - 1.5pkT_p \nabla \ln m_p \quad (929)$$

according to [Equation 64](#) and [Equation 65](#).

For anisotropic mobilities, [Equation 922](#) and [Equation 923](#) must be rewritten as:

$$\vec{J}_n = \mu_n A_{r_n} \vec{g}_n \quad (930)$$

$$\vec{J}_p = \mu_p A_{r_p} \vec{g}_p \quad (931)$$

where  $r_n$  and  $r_p$  are the anisotropy factors for electrons and holes, respectively. If the crystal coordinate system coincides with the simulation coordinate system of Sentaurus Device, depending on the dimension of the problem, the matrices  $A_r$  are given by:

$$A_r = \begin{bmatrix} 1 & & \\ & 1 & \\ & & 1/r \end{bmatrix} \quad \text{or} \quad A_r = \begin{bmatrix} 1 & \\ & 1/r \end{bmatrix} \quad (932)$$

## Chapter 28: Anisotropic Properties

### Anisotropic Mobility

In general, however,  $A_r$  must be written as:

$$A_r = Q \begin{bmatrix} 1 & & \\ & 1 & \\ & & 1/r \end{bmatrix} Q^T \quad \text{or} \quad A_r = Q_{2:2} \begin{bmatrix} 1 & \\ & 1/r \end{bmatrix} Q_{2:2}^T \quad (933)$$

where  $Q$  is defined in [Equation 917–Equation 919](#).

## Driving Forces

In the isotropic case, the electric field parallel to the electron or hole current is given by (see [Equation 363 on page 447](#)):

$$F_c = \frac{\vec{F} \cdot \vec{J}_c}{J_c} = \frac{\vec{F} \cdot \vec{g}_c}{g_c} \quad (934)$$

For anisotropic mobilities:

$$F_c = \frac{\vec{F} \cdot \vec{A} \vec{g}_c}{|\vec{A} \vec{g}_c|} \quad (935)$$

Similarly, the electric field perpendicular to the current, as given in [Equation 337](#), must be rewritten as:

$$F_{c,\perp} = \sqrt{F^2 - \frac{(\vec{F} \cdot \vec{A} \vec{g}_c)^2}{|\vec{A} \vec{g}_c|^2}} \quad (936)$$

[Equation 365](#) shows how the gradient of the Fermi potential  $\Phi_c$  can be used as the driving force in high-field saturation models. For this case, Sentaurus Device has two formulas:

$$F_c = |\vec{A} \nabla \Phi_c| \quad (\text{default formula}) \quad (937)$$

$$F_c = \frac{(\vec{A} \nabla \Phi_c) \cdot \nabla \Phi_c}{|\vec{A} \nabla \Phi_c|} \quad (\text{second formula}) \quad (938)$$

The global Math parameter `AnisoGradQF_formula=0 | 1` selects the corresponding formula.

In the isotropic hydrodynamic Canali model, the driving force  $\vec{F}_c$  satisfies:

$$\vec{F}_c \cdot \mu \vec{F}_c = \frac{w_c - w_0}{\tau_{ec} q} \quad (939)$$

as can be seen from [Equation 369](#). To derive the appropriate expression in the anisotropic case, it is assumed that  $\vec{F}_c$  operates parallel to the current, that is:

$$\vec{F}_c = F_c \hat{e}_c \quad (940)$$

## Chapter 28: Anisotropic Properties

### Anisotropic Mobility

where:

$$\hat{e}_c = \frac{\vec{A}g_c}{|\vec{A}g_c|} \quad (941)$$

is the direction of the electron or hole current.

Instead of [Equation 939](#), you now have:

$$\mu F_c^2 (\hat{A}e_c) \cdot \hat{e}_c = \frac{w_c - w_0}{\tau_{ec} q} \quad (942)$$

or:

$$F_c = \sqrt{\frac{w_c - w_0}{\tau_{ec} q \mu \hat{A} e_c \cdot \hat{e}_c}} \quad (943)$$

---

## Total Anisotropic Mobility

This is the simplest mode in Sentaurus Device. Only a total anisotropy factor  $r_e$  or  $r_h$  is specified in the command file:

```
Physics {
    Aniso(
        eMobilityFactor (Total) = re
        hMobilityFactor (Total) = rh
    )
}
```

Sentaurus Device computes the mobility  $\mu$  for electrons or holes along the main crystallographic axis as usual. The mobility  $\mu_{aniso}$  is then given by:

$$\mu_{aniso} = \frac{\mu}{r} \quad (944)$$

### Note:

In this mode, Sentaurus Device does not update the driving forces as discussed in [Driving Forces on page 895](#).

---

## Self-Consistent Anisotropic Mobility

This is the most accurate, but also the most expensive, mode in Sentaurus Device. The electron and hole mobility models specified in the `Physics` section are evaluated separately for the major and minor crystallographic axes, but with different parameters for each axis.

## Chapter 28: Anisotropic Properties

### Anisotropic Mobility

This option is activated in the `Physics` section for electron or hole mobilities as follows:

```
Physics {
    Aniso(
        eMobility
        hMobility
    )
}
```

To simplify matters, you can specify the following option to activate self-consistent, anisotropic, mobility calculations for both electrons and holes:

```
Physics {
    Aniso( Mobility )
}
```

*Table 151 Isotropic mobility models and their anisotropic versions*

Isotropic mobility model	Anisotropic mobility model
ConstantMobility	ConstantMobility_aniso
DopingDependence	DopingDependence_aniso
EnormalDependence	EnormalDependence_aniso
HighFieldDependence	HighFieldDependence_aniso
UniBoDopingDependence	UniBoDopingDependence_aniso
UniBoEnormalDependence	UniBoEnormalDependence_aniso
UniBoHighFieldDependence	UniBoHighFieldDependence_aniso
HydroHighFieldDependence	HydroHighFieldDependence_aniso

The physical model interface (PMI) also supports anisotropic mobility calculations. The constructors of the classes `PMI_DopingDepMobility`, `PMI_EnormalMobility`, and `PMI_HighFieldMobility` contain an additional flag to distinguish between the isotropic and anisotropic case (see [Chapter 39 on page 1207](#)).

For example, you can specify these parameters for the constant mobility model in the parameter file of Sentaurus Device:

```
ConstantMobility {
    mumax = 1.4170e+03, 4.7050e+02
    Exponent = 2.5, 2.2
}
```

## Chapter 28: Anisotropic Properties

### Anisotropic Metal Resistivity

The following parameters would then compute a reduced constant mobility along the anisotropic axis:

```
ConstantMobility_aniso {  
    mumax = 1.0e+03, 4.0e+02  
    Exponent = 2.5, 2.2  
}
```

In each vertex, Sentaurus Device introduces the anisotropy factors  $r_e$  and  $r_h$  as two additional unknowns. For a given value of  $r$ , the driving forces  $F_c$  and  $F_{c,\perp}$  are computed as discussed in [Driving Forces on page 895](#), and the mobilities along the isotropic and anisotropic axes are obtained. The equation for the unknown factor  $r$  is then given by:

$$r = \frac{\mu(r)}{\mu_{\text{aniso}}(r)} \quad (945)$$

This nonlinear equation is solved in each vertex for both electron and hole mobilities.

---

## Anisotropic Mobility Visualization

You can specify various plot names in the `Plot` section to visualize the anisotropic mobility calculations.

*Table 152 Plot names for anisotropic mobility*

Plot name	Description
eMobility	Electron mobility along main axis
hMobility	Hole mobility along main axis
eMobilityAniso	Electron mobility along anisotropic axis
hMobilityAniso	Hole mobility along anisotropic axis
eMobilityAnisoFactor	Anisotropic factor for electrons
hMobilityAnisoFactor	Anisotropic factor for holes

---

## Anisotropic Metal Resistivity

In thin-metal layered structures, the metal resistivity can differ along different coordinate axes (see [Transport in Metals on page 295](#)).

## Chapter 28: Anisotropic Properties

### Anisotropic Avalanche Generation

#### Note:

The anisotropic metal resistivity option is not supported for conductive insulators (see [Conductive Insulators on page 300](#)).

---

## Total Anisotropic Metal Resistivity Factor

This option is similar to [Total Anisotropic Mobility on page 896](#). Sentaurus Device assumes that the metal resistivity  $\rho$  is the same in all directions perpendicular to the anisotropic direction (see [Aniso\(direction\) Versus LatticeParameters on page 892](#)) and is  $\rho_{\text{aniso}}$  along the direction. The anisotropy factor  $r$  is defined as the ratio:

$$r = \frac{\rho}{\rho_{\text{aniso}}} \quad (946)$$

To activate this option, specify the metal resistivity anisotropy factor  $r$  in the command file regionwise or globally, as follows:

```
Physics {  
    Aniso( MetalResistivityFactor(Total) = r )  
}
```

---

## Self-Consistent Anisotropic Metal Resistivity

This option works similarly to [Self-Consistent Anisotropic Mobility on page 896](#). To set this option, both the `MetalResistivity` and `MetalResistivity_aniso` sections in the parameter file should be used to define the metal resistivity anisotropy and its temperature dependence.

To activate this option, use the following regionwise or global specification in the command file:

```
Physics{  
    Aniso( MetalResistivity )  
}
```

---

## Anisotropic Avalanche Generation

Sentaurus Device computes the avalanche generation according to [Equation 454 on page 494](#). In the isotropic case, the terms  $n v_n$  and  $p v_p$  can also be written, respectively, as:

$$n v_n = \mu_n |g_n| \quad (947)$$

$$p v_p = \mu_p |g_p| \quad (948)$$

## Chapter 28: Anisotropic Properties

### Anisotropic Avalanche Generation

If anisotropic mobilities are switched on, [Equation 947](#) and [Equation 948](#) are replaced by:

$$nv_n = \mu_n |A_{r_n} \vec{g}_n| \quad (949)$$

$$pv_p = \mu_p |A_{r_p} \vec{g}_p| \quad (950)$$

#### Note:

[Equation 949](#) and [Equation 950](#) apply only to total direction-dependent and self-consistent mobility calculations. If you select the total anisotropic option, then [Equation 947](#) and [Equation 948](#) are used (see [Total Anisotropic Mobility on page 896](#)).

Anisotropic avalanche calculations can be activated in the Physics section, independently for electrons and holes:

```
Physics {
    Aniso(
        eAvalanche
        hAvalanche
    )
}
```

The keyword `Avalanche` activates calculations of anisotropic avalanche for both electrons and holes:

```
Physics { Aniso( Avalanche ) }
```

In the anisotropic mode, different avalanche parameters can be specified along the isotropic and anisotropic axes.

*Table 153 Isotropic avalanche models and their anisotropic versions*

Isotropic avalanche model	Anisotropic avalanche model
vanOverstraetendeMan	vanOverstraetendeMan_aniso
Okuto	Okuto_aniso

Sentaurus Device uses interpolation to compute avalanche parameters for an arbitrary direction of the current. Let  $e_c$  be the direction of the electron or hole current as defined in [Equation 941 on page 896](#). In the crystal reference system, the current is given by:

$$\hat{e}'_c = Q^T \hat{e}_c = \begin{bmatrix} e'_x \\ e'_y \\ e'_z \end{bmatrix} \quad (951)$$

## Chapter 28: Anisotropic Properties

### Anisotropic Electrical Permittivity

Sentaurus Device interpolates an avalanche parameter  $p$  depending on the direction of the current  $e_c$ , according to:

$$\begin{aligned} p(\hat{e}_c) &= (e_x^2 + e_y^2) \cdot p_{\text{isotropic}} + e_z^2 \cdot p_{\text{anisotropic}} && \text{for 3D simulations} \\ \hat{p}(e_c) &= e_x^2 \cdot p_{\text{isotropic}} + e_y^2 \cdot p_{\text{anisotropic}} && \text{for 2D simulations} \end{aligned} \quad (952)$$

The PMI also supports the calculation of anisotropic avalanche generation. The current without mobility  $A g_c$  is passed as an input parameter, and it can be used by the PMI code to determine the model parameters depending on the direction of the current (see [Avalanche Generation on page 1259](#)).

#### Note:

The Hatakeyama avalanche model is an anisotropic avalanche model (see [Hatakeyama Avalanche Model on page 505](#)). However, it must not be used with an Aniso(Avalanche) specification.

---

## Anisotropic Electrical Permittivity

The electrical permittivity  $\epsilon$  in [Equation 37](#) can have different values along different crystallographic axes. If the crystallographic axes coincide with the coordinate system of Sentaurus Device, the scalar  $\epsilon$  is replaced by the matrix:

$$E = \begin{bmatrix} \epsilon \\ \epsilon \\ \epsilon_{\text{aniso}} \end{bmatrix} \quad \text{or} \quad E = \begin{bmatrix} \epsilon \\ \epsilon_{\text{aniso}} \end{bmatrix} \quad (953)$$

depending on the dimension of the problem. For general crystallographic axes, the matrix  $E$  is given by:

$$E = Q \begin{bmatrix} \epsilon \\ \epsilon \\ \epsilon_{\text{aniso}} \end{bmatrix} Q^T \quad \text{or} \quad E = Q_{2:2} \begin{bmatrix} \epsilon \\ \epsilon_{\text{aniso}} \end{bmatrix} Q_{2:2}^T \quad (954)$$

where  $Q$  is defined in [Equation 917–Equation 919](#).

Anisotropic electrical permittivity is switched on using the keyword `Poisson` in the `Physics` section of the command file (regionwise or materialwise specification is supported):

```
Physics (Material = "SiC") {
    Aniso (Poisson)
}
```

The model parameters for  $\epsilon$  and  $\epsilon_{\text{aniso}}$  can be specified in the parameter file.

## Chapter 28: Anisotropic Properties

### Anisotropic Thermal Conductivity

[Table 154](#) lists the corresponding models.

*Table 154 Anisotropic electrical permittivity models*

Isotropic model	Anisotropic model
Epsilon	Epsilon_aniso

Different parameters can be specified for each region or each material. The following section in the command file of Sentaurus Device plots electrical permittivities:

```
Plot {  
    DielectricConstant  
    "DielectricConstantAniso"  
}
```

---

## Anisotropic Thermal Conductivity

The thermal conductivity  $\kappa$  in [Equation 71](#) can have different values along different crystallographic axes. If the crystallographic axes coincide with the coordinate system of Sentaurus Device, the scalar  $\kappa$  is replaced by the matrix:

$$K = \begin{bmatrix} \kappa & & \\ & \kappa & \\ & & \kappa_{\text{aniso}} \end{bmatrix} \quad \text{or} \quad K = \begin{bmatrix} \kappa & \\ & \kappa_{\text{aniso}} \end{bmatrix} \quad (955)$$

depending on the dimension of the problem.

For general crystallographic axes, the matrix  $K$  is given by:

$$K = Q \begin{bmatrix} \kappa & & \\ & \kappa & \\ & & \kappa_{\text{aniso}} \end{bmatrix} Q^T \quad \text{or} \quad K = Q_{2:2} \begin{bmatrix} \kappa & \\ & \kappa_{\text{aniso}} \end{bmatrix} Q_{2:2}^T \quad (956)$$

where  $Q$  is defined in [Equation 917–Equation 919](#).

Anisotropic thermal conductivity is switched on using the keyword `Temperature` in the `Physics` section of the command file (regionwise or materialwise specification is supported):

```
Physics(Material = "SiC") {  
    Aniso (Temperature)  
}
```

The model parameters for  $\kappa$  and  $\kappa_{\text{aniso}}$  can be specified in the parameter file.

## Chapter 28: Anisotropic Properties

### Anisotropic Density Gradient Model

[Table 155](#) lists the corresponding models.

*Table 155 Anisotropic thermal conductivity models*

Isotropic model	Anisotropic model
Kappa	Kappa_aniso

Different parameters can be specified for each region or each material. You can also use the PMI to compute anisotropic thermal conductivities. The constructor of the class `PMI_TermalConductivity` has an additional parameter to distinguish between the isotropic and anisotropic directions (see [Thermal Conductivity on page 1350](#)).

The following section in the command file of Sentaurus Device plots thermal conductivities:

```
Plot {
    "ThermalConductivity"
    "ThermalConductivityAniso"
}
```

---

## Anisotropic Density Gradient Model

The density gradient model provides support for anisotropic quantization. See [Density Gradient Model on page 362](#).

---

## Anisotropic Directions for Density Gradient Model

By default, the anisotropic direction of the density gradient model is the same as in all other models. However, the density gradient model can have other anisotropic directions. Moreover, you can have three different axes of anisotropy.

### Example 1

Global anisotropic models and density gradient model have different direction:

```
Physics {
    Aniso(
        # global anisotropic direction relative to crystal coordinate
        # system
        direction(CrystalSystem)=(0.6, 0.8) Poisson Temperature

        # aniso direction for DG model relative to simulation coordinate
        # system
        eQuantumPotential{ direction(SimulationSystem)=(1,1,0) }
    )
}
```

## Chapter 28: Anisotropic Properties

### Anisotropic Density Gradient Model

Parameter file:

```
QuantumPotentialParameters {
    alpha[1] = 4 1      # aniso direction = (1, 1, 0) relative to
                        # simulation coordinate system
    alpha[2] = 1 1      # isotropic value
    alpha[3] = 1 1      # isotropic value, in 3D case alpha[3] = alpha[2]
}
```

### Example 2

Anisotropic density gradient model has three anisotropic axes:

```
Aniso( eQuantumPotential(
    AnisoAxes(SimulationSystem) = {
        ( 0.6, 0.8, 0)
        (-0.8, 0.6, 0)
    }
))
```

Parameter file:

```
QuantumPotentialParameters {
    alpha[1] = 4 1      # aniso axis = ( 0.6, 0.8, 0)
    alpha[2] = 1 1      # aniso axis = (-0.8, 0.6, 0)
    alpha[3] = 2 1      # aniso axis = ( 0.0, 0.0, 1) this vector is computed
}
```

## Ferroelectric Materials

---

*This chapter explains how ferroelectric materials are treated in a simulation using Sentaurus Device.*

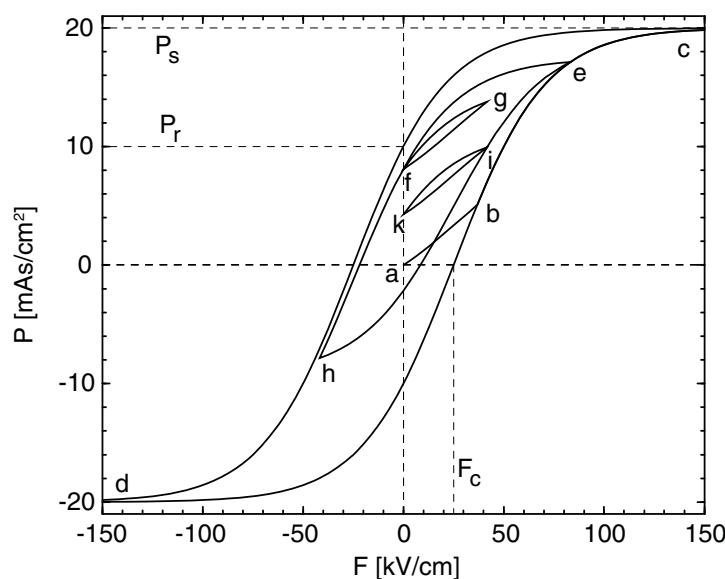
In ferroelectric materials, the polarization  $\vec{P}$  depends nonlinearly on the electric field  $\vec{F}$ . The polarization at a given time depends on the electric field at that time and the electric field at previous times. The history dependence leads to the well-known phenomenon of hysteresis, which is used in nonvolatile memory technology.

---

### Using Ferroelectrics

Sentaurus Device implements a model for ferroelectrics that features minor loop nesting and memory wipeout. [Figure 55](#) demonstrates these properties and [Ferroelectrics Model on page 908](#) discusses them further.

*Figure 55 Example polarization curve*



## Chapter 29: Ferroelectric Materials

### Using Ferroelectrics

To activate the model, specify the keyword `Polarization` in the `Physics` section of the command file. Use the optional parameter `Memory` to prescribe the maximum-allowed nesting depth of minor loops. The smallest allowed value for `Memory` is 2; the default value is 10. If minor loop nesting becomes too deep, then the nesting property of the minor loops can be lost. However, the polarization curve remains continuous.

The following example switches on the ferroelectric model in region `Region.17` and sets the size of the memory to 20 turning points for each element and each mesh axis:

```
Physics (region = "Region.17") {
    Polarization (Memory=20)
}
```

To obtain a plot of the polarization field, specify `Polarization/Vector` in the `Plot` section of the command file.

Sentaurus Device characterizes the static properties of a ferroelectric material by three parameters: the remanent polarization  $P_r$ , the saturation polarization  $P_s$ , and the coercive field  $F_c$ . The hysteresis curve in [Figure 55 on page 905](#) illustrates these quantities. Furthermore, Sentaurus Device parameterizes the transient response of the ferroelectric material by the relaxation times  $\tau_E$  and  $\tau_P$ , and by a nonlinear coupling constant  $k_n$  (see [Ferroelectrics Model on page 908](#)).

Specify the values for these parameters in the `Polarization` parameter set. For example:

```
Polarization
{
    * Remanent polarization P_r, saturation polarization P_s,
    * and coercive field F_c for x,y,z direction (crystal axes)
    P_r = (1.0000e-05, 1.0000e-05, 1.0000e-05) #[C/cm^2]
    P_s = (2.0000e-05, 2.0000e-05, 2.0000e-05) #[C/cm^2]
    F_c = (2.5000e+04, 2.5000e+04, 2.5000e+04) #[V/cm]
    * Relaxation time for the auxiliary field tau_E, relaxation
    * time for the polarization tau_P, nonlinear coupling kn.
    tau_E = (0.0000e+00, 0.0000e+00, 0.0000e+00) #[s]
    tau_P = (0.0000e+00, 0.0000e+00, 0.0000e+00) #[s]
    kn     = (0.0000e+00, 0.0000e+00, 0.0000e+00) #[cm*s/V]
}
```

The parameters in this example are the defaults for the material `Insulatorx`. For all other materials, all default values are zero. Each of the three numbers given for any of the parameters corresponds to the value for the respective coordinate axis of the mesh. If a `P_s` component is zero, then the ferroelectric model is deactivated along the corresponding direction. If a `P_s` component is nonzero, then the respective `P_r` and `F_c` components must also be nonzero. Furthermore, the `P_r` component must be smaller than the `P_s` component. By default, the relaxation times are zero, which means that polarization follows the applied electric field instantaneously.

To start the simulation from a given polarization state  $P_{ini}$ , use the optional keyword `initialvector` to prescribe the components of the initial polarization vector. The absolute value of each component given for `initialvector` must be smaller than or equal to the

## Chapter 29: Ferroelectric Materials

### Using Ferroelectrics

remnant polarization `P_x`, along the corresponding axis. The default for `initialvector` is a null vector.

The following example sets the components of the initial polarization vector in region `R.Ferro` to  $1.0e-05 \text{ C/cm}^2$  along each axis of the mesh:

```
Physics (Region= "R.Ferro") {
    Polarization (initialvector= (1.0e-05 1.0e-05 1.0e-05))
}
```

In devices with ferroelectric and semiconductor regions, it might be difficult to obtain an initial solution of the Poisson equation. In many cases, the `LineSearchDamping` keyword can solve these problems (see [Damped Newton Iterations on page 195](#)). To use this keyword, start the simulation as follows:

```
coupled (LineSearchDamping=0.01) { Poisson }
```

Due to the decoupled handling of the polarization vector components (see [Ferroelectrics Model](#)) and the nonlinearity of the polarization versus electric-field relation, the resulting polarization profiles might not reflect the device symmetry for nonrectangular structures. Under these circumstances, it is helpful to specify the polarization direction different from the default ones, aligned with the coordinate axes. This is achieved with the help of the predefined reference surfaces.

In this case, the direction of the polarization vector at a given point in space is taken to be opposite to the geometrically shortest path connecting this point and the closest point on the specified surface. Such a vector is normal to the reference surface. This allows the polarization vector field to follow the device symmetry. As an example, for a cylindrical geometry, if the surface is taken as an exterior of the cylinder, then the polarization will align along the lines connecting the external surface and the center of the cylinder.

The following example illustrates how to use this feature in a simulation scenario where the polarization in region `"R.Ferro"` is perpendicular to the electrode `"C.Gate"`, which composes the `"S.Gate"` surface:

```
Electrode { {Name="C.Gate" ...} }

Math { Surface "S.Gate" (Electrode="C.Gate") }

Physics (Region="R.Ferro") {
    Polarization(SurfaceName="S.Gate" SurfaceReconstruction)
}
```

The `Surface` defined in the global `Math` section can be a union of an arbitrary number of region or material interfaces and electrodes (see [Table 211 on page 1573](#)).

When performing simulations of ferroelectric gate-all-around structures discretized with an axis-aligned mesh, it is recommended to use the reference surface specified by the `SurfaceName` keyword together with the `SurfaceReconstruction` option, as in the previous example. This option activates an improved algorithm for calculating the directional

polarization. This combination alleviates the impact of device geometry and mesh quality on the physical results and obtains reliable polarization profiles independently on the mesh.

## Ferroelectrics Model

The vector quantity  $\vec{P}$  is split into its components along the main axes of the mesh coordinate system. This results in one to three scalar problems. Sentaurus Device handles each problem separately using the model from [1] with extensions for transient behavior [2]. The implemented approach can be recognized as a Preisach-based model of ferroelectric hysteresis.

First, Sentaurus Device computes an auxiliary field  $F_{\text{aux}}$  from the electric field  $F$ :

$$\frac{d}{dt}F_{\text{aux}}(t) = \frac{F(t) - F_{\text{aux}}(t)}{\tau_E} \quad (957)$$

Here,  $\tau_E$  is a material-specific time constant. For  $\tau_E = 0$  or for quasistationary simulations,  $F_{\text{aux}} = F$ .

From the auxiliary field, Sentaurus Device computes the auxiliary polarization  $P_{\text{aux}}$ . The auxiliary polarization  $P_{\text{aux}}$  is an algebraic function of the auxiliary field  $F_{\text{aux}}$ :

$$P_{\text{aux}} = c \cdot P_s \cdot \tanh(w \cdot (F_{\text{aux}} \pm F_c)) + P_{\text{off}} \quad (958)$$

where  $P_s$  is the saturation polarization,  $F_c$  is the coercive field, and:

$$w = \frac{1}{2F_c} \ln \frac{P_s + P_r}{P_s - P_r} \quad (959)$$

where  $P_r$  is the remanent polarization. In Equation 958, the plus sign applies to the decreasing auxiliary field and the minus sign applies to the increasing auxiliary field. The different signs reflect the hysteretic behavior of the material.  $P_{\text{off}}$  and  $c$  in Equation 958 result from the polarization history of the material, see below.

Finally, from the auxiliary polarization and auxiliary field, Sentaurus Device computes the actual polarization  $P$ :

$$\frac{d}{dt}P(t) = \frac{P_{\text{aux}}[F_{\text{aux}}(t)] - P(t)}{\tau_P} \cdot \frac{1 + k_n | \frac{d}{dt}F_{\text{aux}}(t) |}{1 + k_n | \frac{d}{dt}F_{\text{aux}}(t) |} \quad (960)$$

Here,  $\tau_P$  and  $k_n$  are material-specific constants. For  $\tau_P = 0$  or for quasistationary simulations,  $P = P_{\text{aux}}$ .

Upper and lower turning points are points in the  $P_{\text{aux}} - F_{\text{aux}}$  diagram where the sweep direction of the auxiliary field  $F_{\text{aux}}$  changes from increasing to decreasing, and from decreasing to increasing, respectively. At each bias point, the most recent upper and lower turning points,  $(F_u, P_u)$  and  $(F_l, P_l)$ , must both be on each of the two curves defined by Equation 958; this requirement determines  $P_{\text{off}}$  and  $c$ .

## Chapter 29: Ferroelectric Materials

### Ferroelectrics Model

Sentaurus Device *memorizes* turning points as they are encountered during a simulation. The memory always contains  $(\infty, P_s)$  as the oldest and  $(-\infty, -P_s)$  as the second-oldest turning point. By using [Equation 958](#), these two points define a pair of curves with  $c = 1$  and  $P_{\text{off}} = 0$ ; together, the two curves form the saturation loop. All other pairs of turning points result in  $c < 1$  and define a pair of curves forming minor loops.

When the auxiliary field leaves the interval defined by  $F_l$  and  $F_u$  of the two newest turning points, these two turning points are removed from the memory; this reflects the memory wipeout observed in experiments. The pair of turning points that are newest in the memory, after this removal, determines the further  $P_{\text{aux}}(F_{\text{aux}})$  relationship.

As the older of the dropped turning points was originally reached by walking on the curve defined by the turning points that now (after dropping) again determine  $P_{\text{aux}}(F_{\text{aux}})$ , the polarization curve remains continuous. For example, see points e, f, and i in [Figure 55 on page 905](#). Furthermore, the minor loop defined by the two dropped turning points is nested inside the minor loop defined by the present turning points. The nesting of minor loops is also a feature known from experiments on ferroelectrics. [Figure 55](#) illustrates this by the loops f-g and i-k, both of which are nested in loop e-h, which in turn is nested in loop c-d.

In small-signal (AC) analysis, a very small periodic signal is added to the DC bias (see [Small-Signal AC Analysis on page 149](#)). As a result, the (auxiliary) polarization at each point of the ferroelectric material changes along a very small minor loop nested in the main loop that stems from the DC variation of the bias voltage. The average slope of this minor loop is always smaller than the slope of the main loop at the point where the loops touch. Consequently, even at very low frequencies, the AC response of the system is different from what would be obtained by taking the derivative of the DC curves.

You can instruct Sentaurus Device to stay on the main branch of the hysteresis loop when computing the AC response. To this end, specify `MainLoopAC` to the `Polarization` keyword as in the following example:

```
Physics (Region = "R.Ferro") { Polarization (MainLoopAC) }
```

This gives rise to multivalued butterfly-like C–V characteristics when calculating the AC response of ferroelectric materials. At low frequencies, the AC response calculated in this way closely resembles the derivative of the DC curves.

As an example of how the turning point memory works, see [Figure 55](#). The points on the polarization curve are reached in the sequence a, b, c, d, e, f, g, f, h, i, k, i, e, c. For simplicity, a quasistationary process is assumed and, therefore,  $F_{\text{aux}} = F$  and  $P_{\text{aux}} = P$ . Starting the simulation at point a, the newest point in memory is b (Sentaurus Device initializes it in this way). The second newest point is a negative saturation point (l), and the oldest point is a positive saturation point (u). This memory state is denoted by [blu]. For this state, a is on the decreasing field curve. The curve segment (that is, the coefficients  $c$  and  $P_{\text{off}}$ ) from a to b is determined by the points l and b.

Ramping up from a makes a a turning point, so the memory becomes [ablu]. When crossing b and proceeding to c, a and b are dropped from the memory. Therefore, the memory

becomes [lu]. These two points determine the curve from b to c. Turning at c, c is added to the memory, giving [clu]. From c to d, use c and l; at d, the memory becomes [dclu]; at point e, [edclu]; at f, [fedclu]; at g, [gfedclu]. Passing through f, the two newest points, f and g, are dropped and the memory is [edclu]; at h, [hedclu]; at i, [ihedclu]; at k, [kihedclu]. At i again, i and k are dropped, giving [hedclu]; at e, e and h are dropped, giving [dclu].

At the beginning of a simulation, the memory contains one turning point chosen such that the point  $E_{aux} = 0$ ,  $P_{aux} = P_{ini}$  is on the minor loop so defined (for example, point b in [Figure 55 on page 905](#), in the case of  $P_{ini} = 0$ ). The nature of the model is such that it is not possible to have a state of the system that is completely symmetric. In particular, even for a symmetric device and at the very beginning of the simulation,  $P(E) \neq -P(-E)$ . This asymmetry is most prominent for the virgin curves (for example, a-b in [Figure 55](#)) of the ferroelectric, which are different for different signs of voltage ramping.

## Ginzburg–Landau–Khalatnikov Equation

The Ginzburg–Landau–Khalatnikov equation provides a reliable description of ferroelectric material properties in terms of a free energy  $F$  expanded as a power series in the ferroelectric polarization  $P$ . The state of the system is determined by minimizing this free energy  $F(P)$  with respect to  $P$ . Interesting properties of ferroelectric materials emerge from a characteristic multi-well free-energy polarization landscape. Switching between the metastable minima of  $F(P)$  due to an applied electric field  $E$  gives rise to an electric-field polarization hysteresis phenomenon, used in nonvolatile memory technology. Stabilization of the saddle point of  $F(P)$  by a depolarizing electric field allows for ferroelectric negative-capacitance operation, which is used in low-power computing applications.

Based solely on symmetry considerations, the Landau theory provides an expansion of the free energy  $F$  as a power series of the order parameter [\[3\]\[4\]](#). For ferroelectric phase transitions, this order parameter is the polarization vector  $\mathbf{P}$ . The free energy of a ferroelectric material occupying a region is written as [\[3\]\[4\]](#):

$$F = \underbrace{\alpha_i P_i^2 + \beta_i P_i^4 + \gamma_i P_i^6}_{\Omega} - g_{ij} \frac{\partial P_i}{\partial x_j} - E_i P_i + \epsilon_0 E_i E_i d\Omega \quad (961)$$

In the Ginzburg–Landau–Khalatnikov framework, the governing equation of polarization evolution can be obtained as a gradient flow associated with the free energy functional given in [Equation 961](#):

$$\rho \frac{dP_i}{dt} + \nabla_{P_i} F = 0 \quad (962)$$

From [Equation 961](#) and [Equation 962](#), the electric field–polarization relation can be derived:

$$\rho \frac{dP_i}{dt} + 2\alpha_i P_i + 4\beta_i P_i^3 + 6\gamma_i P_i^5 - 2g_{ij} \frac{\partial^2 P_i}{\partial x_j^2} - E_i = 0 \quad (963)$$

## Chapter 29: Ferroelectric Materials

### Ginzburg–Landau–Khalatnikov Equation

In these equations,  $P_i$  and  $E_i$  denote the Cartesian components ( $ij = x, y, z$ ) of the polarization vector  $\mathbf{P}$  in C/cm<sup>2</sup> and the electric field vector  $\mathbf{E}$  in V/cm, respectively.

Each term in [Equation 961](#) has a physical interpretation:

- The first three terms correspond to the stored Landau energy due to polarization. The Landau energy builds in the information that the crystal prefers certain spontaneous polarization at its minima. The Landau coefficients  $\alpha_i$ ,  $\beta_i$ , and  $\gamma_i$  are used to parameterize the energy function, which can be extracted from electric-field polarization measurements of a ferroelectric capacitor.
- The fourth term, associated with polarization gradients, accounts for the potential energy caused by nonuniform polarization and is regarded as the energetic cost of forming domain walls [3]. This term penalizes sharp changes in the spatial distribution of the polarization vector field. The width of the polarization domain walls is proportional to the square root of the smallest eigenvalue of the matrix composed from the  $g_{ij}$  coefficients.
- If the polarization deviates from the preferred states, corresponding to the minima of  $F[\mathbf{P}]$ , the energy cost must be compensated by the potential energy of the applied electric field  $\mathbf{E}$  as described by the fifth term. This term enforces the need of the polarization to align with the electric field.
- The last term is the electrostatic energy associated with the electric field that is generated by the polarization distribution. For any polarization distribution, the electrostatic potential is determined by solving the Poisson equation (see [Equation 37](#)) in the entire simulation domain, subject to appropriate boundary conditions.

The steady-state solution ( $dP_i/dt = 0$ ) of [Equation 963](#) corresponds to a thermal equilibrium state of the ferroelectric. In this state, the free energy attains its minimum when [Equation 961](#) is optimized with respect to the polarization distribution under the applied electric field. The equilibrium polarization distribution cannot be arbitrary. Instead, the energy-minimized polarization vector field must satisfy the equation of continuity [5]:

$$\frac{\partial P_i}{\partial x_i} = 0 \quad (964)$$

The electrical compatibility condition expressed in [Equation 964](#) ensures that polarization is divergence free and, consequently, the ferroelectric domain remains charge free. Therefore, the divergence-free polarization distribution is energy minimized.

---

## Using the Ginzburg–Landau–Khalatnikov Equation

To activate the equation, you must specify the keyword `FEPolarization` in the regionwise or materialwise `Physics` sections of the command file. Since ferroelectrics are insulators, the equation can be activated only for the materials of group `Insulator` (see [User-Defined Materials on page 63](#)).

## Chapter 29: Ferroelectric Materials

### Ginzburg–Landau–Khalatnikov Equation

Apart from activating the equation in the Physics section, the equation must be solved by adding `FEPolarization` in the `Solve` section. In general, it is recommended to perform transient simulations when using this equation. This emulates the physical process of polarization evolution and allows you to verify the decrease property of the free energy along the solution paths, for convergence to a physically relevant solution.

The following example solves the equation in region `GateOxide`. First, the Poisson equation is solved alone to obtain an initial guess of the electrostatic potential. Second, a transient simulation is performed for 1 s to find an initial state of the device. This is followed by biasing the gate contact from its initial value at t=1 s to 1 V at t=2 s by using a transient ramp command (see [Transient Ramps on page 144](#)):

```
Physics ( Region="GateOxide" ) {
    FEPolarization
}
Solve {
    Poisson
    Transient (
        InitialTime=0 FinalTime=1
    ) { Coupled { Poisson FEPolarization } }
    Transient (
        InitialTime=1 FinalTime=2
        Goal { name="Gate" Voltage=1 }
    ) { Coupled { Poisson FEPolarization } }
}
```

You can plot the polarization vector field by adding, in the `Plot` section, either of the following keywords:

- `FEPolarization/Vector` for plotting polarization vector distributions on vertices
- `FEPolarization/Element/Vector` for element-based plots (see [Device Plots on page 177](#))

For example, the following statement saves the `FEPolarization` vertex-vector dataset to the output file:

```
Plot { FEPolarization/Vector }
```

The model parameters are as follows:

- $\alpha_i$ ,  $\beta_i$ , and  $\gamma_i$  are ferroelectric material parameters, typically extracted from measurements.
- $\rho$  is the viscosity, a kinetic coefficient associated with polarization-switching dynamics.
- $g_{ij}$  is a coupling coefficient for the polarization gradient term of the free energy.

These parameters can be accessed in the parameter file and can be specified regionwise or materialwise. The ferroelectric property is considered isotropic by default. Therefore, the

## Chapter 29: Ferroelectric Materials

### Ginzburg–Landau–Khalatnikov Equation

multiple coefficients in [Equation 963](#) can be replaced by single scalar values:  $\alpha_i$ =alpha,  $\beta_i$ =beta,  $\gamma_i$ =gamma, and  $g_{ij}$ =g.

For example:

```
FEPolarization {
    alpha = -1.3e+10      # cm/F
    beta  = 1.3e+20        # cm^5/(FC^2)
    gamma = 0.0            # cm^9/(FC^4)
    g     = 1.0e-03        # cm^3/F
    rho   = 1.0e+04        # Ohm*cm
}
```

This example corresponds to a ferroelectric undergoing a second-order (continuous) phase transition, since  $\alpha < 0$  and  $\beta > 0$ . For first-order (discontinuous) phase transitions, set  $\alpha < 0$ ,  $\beta < 0$ , and  $\gamma > 0$ . In general,  $\beta$ ,  $\gamma$ , or both coefficients must be set to a positive value. Typical values of  $g$  range from  $10^{-6}$  to  $10^{-2}$   $\text{cm}^3/\text{F}$ . Mole fraction dependency is not supported for these parameters.

*Table 156 Parameters and default values for Ginzburg–Landau–Khalatnikov equation*

Name	Description	Default value	Unit
alpha	$\alpha$ parameter	-4e9	cm/F
beta	$\beta$ parameter	-5e16	cm <sup>5</sup> /(FC <sup>2</sup> )
gamma	$\gamma$ parameter	5e25	cm <sup>9</sup> /(FC <sup>4</sup> )
g	Gradient term	1e-3	cm <sup>3</sup> /F
rho	$\rho$ viscosity	2.25e4	$\Omega\text{cm}$

The quality of the divergence-free polarization constraint, as expressed by [Equation 964](#), is controlled by the `FEPolarizationIP` parameter, which is used in the global `Math` section:

```
Math { FEPolarizationIP=1.0 }
```

This value applies to all regions where the Ginzburg–Landau–Khalatnikov equation (`FEPolarization`) is activated. Generally, the higher the value of `FEPolarizationIP`, the better the approximation of the divergence-free condition on the polarization vector field. However, too high values of `FEPolarizationIP` might overshadow the equation of state, leading to spurious simulation results. Therefore, take care when adjusting this parameter. The recommended values are of order O(1). The default value of `FEPolarizationIP` is zero.

## Chapter 29: Ferroelectric Materials

### Ginzburg–Landau–Khalatnikov Equation

The anisotropic properties of a ferroelectric material can be modeled by referring to the `FEPolarization_tensor` parameter set. To use it, specify `FEPolarization(Tensor)` in the `Physics` section, when activating the model. In this parameter set, each of the elements of the Landau coefficients,  $\alpha_i = \text{alphai}$ ,  $\beta_i = \text{betai}$ , and  $\gamma_i = \text{gammai}$ , corresponds to the  $P_i$  polarization component along the respective coordinate axis of the mesh, where  $i = 1, 2, 3$ . Analogously, the  $g_{ij}$  element of the anisotropic polarization gradient factor affects the spatial derivative of the  $P_i$  polarization component along the  $x_j$  direction, where  $i, j = 1, 2, 3$ . Generally, the polarization gradient coefficients  $g_{ij}$  form a symmetric  $3 \times 3$  tensor. Due to symmetry, it has only six independent components, and it is convenient to express it in a contracted six-component vector notation (Voigt form):

$$g_{11} = g1, g_{22} = g2, g_{33} = g3, g_{23} = g_{32} = g4, g_{13} = g_{31} = g5, g_{12} = g_{21} = g6 \quad (965)$$

which is employed in the `FEPolarization_tensor` parameter set.

In the following example, a material is simulated that displays ferroelectric properties only along the x-axis and y-axis of the coordinate system. In the z-direction of the mesh, it behaves like a linear dielectric:

```
FEPolarization_tensor {  
    alphai = -1.3e+10 # [cm/F]  
    alpha2 = -1.3e+10 # [cm/F]  
    alpha3 = 1.3e+12 # [cm/F]  
    beta1 = 1.3e+20 # [cm^5/(FC^2)]  
    beta2 = 1.3e+20 # [cm^5/(FC^2)]  
    beta3 = 0.0 # [cm^5/(FC^2)]  
    gamma1 = 0.0 # [cm^9/(FC^4)]  
    gamma2 = 0.0 # [cm^9/(FC^4)]  
    gamma3 = 0.0 # [cm^9/(FC^4)]  
    g1 = 1.0e-03 # [cm^3/F]  
    g2 = 1.0e-03 # [cm^3/F]  
    g3 = 1.0e-03 # [cm^3/F]  
    g4 = 1.0e-03 # [cm^3/F]  
    g5 = 1.0e-03 # [cm^3/F]  
    g6 = 1.0e-03 # [cm^3/F]  
    rho = 1.0e+04 # [Ohm*cm]  
}
```

Such an anisotropic ferroelectric material can be successfully integrated in a 3D FinFET device with the z ( $i = 3$ )-axis of the mesh assumed to be a transport direction. In this case, the ferroelectric field effect is caused by the x- and y-polarization components, perpendicular to the channel. The fact that no ferroelectric effect is present along the channel has a negligible impact on the device operation.

By default, all of the polarization vector components are initialized to zero. You can use the `SFactor` parameter to specify a dataset name (which includes the PMI user fields from `PMIUserField0` through to `PMIUserField299`) from which the vertex values of the initial polarization vector component will be initialized. If a dataset name is not specified, then the corresponding vector component is set to zero. The following example initializes the

y-component of the ferroelectric polarization vector from the `PMIUserField1` dataset in region `FE1`:

```
Physics (Region="FE1") {
    FEPolarization( sFactor=( "", "PMIUserField1", "" ) )
}
```

Alternatively, the `SFactor` values can be specified in the parameter file, which results in a uniform initial polarization:

```
Region = "FE1" {
    FEPolarization_tensor {
        ...
        sfactor1 = 0.      # x component [C/cm^2]
        sfactor2 = 1e-6    # y component [C/cm^2]
        sfactor3 = 0.      # z component [C/cm^2]
    }
}
```

---

## Versions of the Solver

Different solver versions are available:

- [Default](#)
- [Multidimensional](#)
- [Solver II](#)

## Default

For the default `FEPolarization` solver (first available in Version O-2018.06), the divergence-free polarization constraint ([Equation 964](#)) is not supported. Consequently, the `FEPolarizationIP` parameter is ignored.

In addition, you must specify `direction` along the axis in the `FEPolarization` statement of the `Physics` section. For example:

```
Physics (Region="GateOxide") {
    FEPolarization ( direction="x" )
}
```

Here, `x` defines the switchable polarization component that is solved. The `direction` parameter takes only `x`, `y`, or `z` as its value. Polarization can be solved along the specified direction, which is the direction perpendicular to the gate, given the fact that the polarization vector can switch only along this direction. To solve tri-gate structures or FinFETs, you can define different ferroelectric regions and specify the appropriate direction in each region.

**Note:**

You can define the model parameters in either the command file or the parameter file, except for `direction` that can be defined only in the command file.

When using the parameter file, you use the `FEPolarization` parameter set (see [Table 156 on page 913](#)), which is the same as the default implementation, except that mole fraction-dependent parameters for compound materials are supported in this version (see [Parameters for Composition-Dependent Materials on page 72](#)). However, it is not possible to use the `FEPolarization_tensor` parameter set with this solver.

**Note:**

When you define parameters in both the command file and the parameter file, those in the command file take precedence. In addition, all parameters must be defined entirely in the command file or the parameter file. Mixed definitions are not allowed.

Since this version solves only one component of the polarization vector field, the output is stored in a scalar dataset `FEPolarization` instead of a vector dataset.

To plot `FEPolarization`, specify:

```
Plot { FEPolarization }
```

By default, the polarization field is initialized to zero. You can use `sFactor` to set the initial value of the polarization. The following example initializes the polarization field  $P_x$  to the value stored in `PMIUserField1`:

```
Physics (Region="GateOxide") {
    FEPolarization ( direction="x" sFactor=( "PMIUserField1" ) )
}
```

## Multidimensional

To solve simultaneously for all of the Cartesian components of the polarization vector field, it is recommended to use this version of the `FEPolarization` solver, which is activated by specifying the `CompatibleFEPolarization` keyword in the global `Math` section. This version supports the `FEPolarizationIP` parameter for simulating solenoidal (divergence-free) polarization vector fields as in the following example:

```
Math {
    CompatibleFEPolarization
    FEPolarizationIP=1.0
}
```

## Chapter 29: Ferroelectric Materials

### Ginzburg–Landau–Khalatnikov Equation

You must also set `direction="all"` in the `FEPolarization` statement of the `Physics` section as follows:

```
Physics {
    FEPolarization ( direction="all" )
}
```

When using `CompatibleFEPolarization`, special care must be taken if the ferroelectric region is not rectangular. You might observe a nonphysical diminishing of the polarization close to nonaxis-aligned interfaces. This issue can be alleviated by lowering the value of `FEPolarizationIP`. However, this might result in a poor approximation of the polarization divergence-free condition, which has a negative impact on numeric stability. For structures with nonrectangular regions, it is recommended to use the default `FEPolarization` approach.

This version supports the `FEPolarizationIP` parameter for simulating solenoidal (divergence-free) polarization vector fields as in the following example:

```
Math {
    CompatibleFEPolarization
    FEPolarizationIP=1.0
}
```

The `FEPolarization_tensor` parameter set is not supported in this version.

## Solver II

The previously discussed geometric restriction can be removed by using the new version of the solver activated by specifying the `FEPolarizationSolver2` keyword in the global `Math` section. This solver supports the `FEPolarizationIP` parameter and the parameter set `FEPolarization_tensor`, and it is the most stable solver.

However, due to numerics and software-related aspects, the following physical models do not yet work with the Ginzburg–Landau–Khalatnikov equation:

- [Dipole Layer on page 230](#)
- [Thermodynamic Model for Current Densities on page 240](#)
- [Hydrodynamic Model for Current Densities on page 241](#)
- Temperature equations (see Chapter 9 on page 245)
- [Modified Ohmic Contacts on page 259](#)
- [Contacts on Insulators on page 260](#)
- [Floating Contacts on page 280](#)
- [Periodic Boundary Conditions on page 284](#)

## Chapter 29: Ferroelectric Materials

### Ginzburg–Landau–Khalatnikov Equation

- [Explicit Trap Occupation on page 550](#)
- [Diffusion of Hydrogen in Oxide on page 603](#)
- [Degradation in Insulators on page 609](#)
- Tunneling (see [Chapter 24 on page 819](#))
- [Ferroelectrics on page 1413](#)

In addition, not all numeric techniques can be used with the Ginzburg–Landau–Khalatnikov equation. The following methods are not supported:

- Mixed-mode simulations (see [Chapter 3 on page 90](#))
- Small-Signal AC Analysis on [page 149](#)
- Harmonic Balance on [page 153](#)
- AC Simulation on [page 1191](#)
- Harmonic Balance Analysis on [page 1194](#)
- TRBDF Composite Method on [page 1200](#)

---

## Bitlis Solver

A specialized linear solver is available for electrostatic problems, including the Poisson and FEPolarization equations. You activate this solver by specifying `Bitlis` as an option to the `Method` keyword, which is used in Sentaurus Device to select various solution algorithms (see [Linear Solvers on page 201](#) for details about the `Method` keyword).

The iterative linear solver Bitlis implements the generalized minimal residual (GMRES) method. The parameters of this solver are specified in parentheses after the solver declaration in the global `Math` section as follows:

```
Math {
    Method = Bitlis (Restart=100, Tolerance=1e-5, Iterations=200)
}
```

The parameter `Restart` determines a number of GMRES back-vectors. If the solution did not converge within a given `Restart` number, then GMRES restarts the iteration. A higher number of back-vectors improves convergence but at the expense of higher memory and execution time. The `Tolerance` coefficient controls the stopping criterion of the iterative method. The method stops if the norm of the residual is reduced by this factor or if a maximum number of iterations is reached. The latter is specified by the `Iterations` parameter. [Table 157](#) lists the default values of these parameters.

*Table 157 Parameters of Bitlis solver*

Parameter	Default value
Iterations	200
Restart	100
Tolerance	1e-8

The Bitlis solver can be combined with a multi-grid preconditioner to achieve good performance and to reduce the time to compute the solution. A multi-grid method tries to approximate the original discrete equations on a hierarchy of grids and uses solutions from coarse grids to accelerate the convergence on the finest grid. You activate the multi-grid preconditioner in Sentaurus Device by specifying the `MGPreconditioner` statement in the global `Math` section. This statement accepts multiple parameters that control the multi-grid algorithm:

```
Math {
    MGPreconditioner(
        MaxLevels=3,
        MinSize=20,
        RelaxationError=1e-10,
        MaxIterations=3,
        SolutionError=1e-8,
        MaxCycles=3
    )
}
```

The parameter `MaxLevels` allows you to declare a maximum number of interpolation levels that build the multi-grid hierarchy in Sentaurus Device. The parameter `MinSize` alters the multi-grid hierarchy and can be used to specify the minimum size of the coarsest grid. Sentaurus Device will not create further interpolation levels if current level size is less than this number.

On the coarsest level, the direct linear solver PARDISO is invoked to solve an approximate discrete problem. The solution of this problem is interpolated to the finest grid using multi-grid techniques. In this process, smoothing of the error is necessary. To this end, a few iterations of the basic iterative damped Jacobi method are invoked. The parameter `RelaxationError` controls the smoothness of the error as it declares the convergence criterion for the smoother. That is, the smoothing iteration stops if the norm of the preconditioned residual is reduced by this factor. The maximum number of smoother sweeps is set by the `MaxIterations` parameter. Better convergence and a larger number of smoothing iterations improve the quality of the approximate solution within the multi-grid preconditioner. However, the application time of the multi-grid preconditioner in every GMRES iteration increases in this case.

## Chapter 29: Ferroelectric Materials

### References

The parameters `SolutionError` and `MaxCycles` control the overall convergence of the multi-grid preconditioner.

Only the solutions of the Poisson and `FEPolarization` equations are accelerated with the multi-grid preconditioner for the Bitlis iterative method. Other equations are solved using PARDISO (see [Linear Solvers on page 201](#) for details about this method).

**Note:**

If you specify the `MGPreconditioner` statement in the `Math` section, but the Bitlis solver is not used, then the statement has no effect on the linear solve. The Bitlis solver is not compatible with the `Blocked` method.

---

## References

- [1] B. Jiang *et al.*, “Computationally Efficient Ferroelectric Capacitor Model for Circuit Simulation,” in *Symposium on VLSI Technology*, Kyoto, Japan, pp. 141–142, June 1997.
- [2] K. Dragosits, *Modeling and Simulation of Ferroelectric Devices*, PhD thesis, Technische Universität Wien, Vienna, Austria, December 2000.
- [3] P. Chandra and P. B. Littlewood, “A Landau Primer for Ferroelectrics,” *Topics in Applied Physics*, vol. 105, pp. 69–116, 2007.
- [4] N. Ng, R. Ahluwalia, and D. J. Srolovitz, “Depletion-layer-induced size effects in ferroelectric thin films: A Ginzburg-Landau model study,” *Physical Review B*, vol. 86, p. 094104, September 2012.
- [5] L. D. Landau and I. M. Khalatnikov, “On the Anomalous Absorption of Sound Near a Second Order Phase Transition Point” (*Dokl. Akad. Nauk SSSR*, 96, 469, 1954), *Collected Papers of L. D. Landau*, D. Ter Haar (ed.), Oxford: Pergamon, pp. 626–629, 1965.

# 30

## Ferromagnetism and Spin Transport

---

*This chapter introduces the physics of spin transfer torque (STT) devices and describes the models used to simulate their behavior in Sentaurus Device.*

---

### Brief Introduction to Spintronics

Conventional semiconductor devices are based on charge transport. Electrons are treated as charged particles whose motion gives rise to current flow. In addition to their charge, electrons carry an intrinsic angular momentum called the electron *spin*. A charged particle with angular momentum acts as a magnetic dipole. In the absence of spatial ordering of the individual magnetic moments, their magnetic fields cancel out, and the effect of spin on electronic transport is small.

In ferromagnetic materials, however, the net magnetic moments of the inner electrons (typically, d- or f-orbitals) at neighboring lattice sites are aligned in parallel by the *exchange interaction*: Extended magnetic *domains* form, and the resulting magnetic field gives rise to a large difference in the energy of the state of conduction electrons depending on the relative orientation of magnetization and the spin direction.

The interaction between the spin of conduction electrons and the magnetization of ferromagnetic regions incorporated into the device structure gives rise to a whole class of *spintronics* devices. For example, the tunneling current between two ferromagnetic regions separated by a thin insulator becomes a function of the angle between the magnetization directions on either side of the barrier (the tunneling magnetoresistance effect). Conversely, the flow of spin-polarized electrons is accompanied by the transport of angular momentum (the spin current), and the absorption of spin current in a ferromagnetic region might change the magnetization direction (the spin transfer torque effect) [1].

The following sections describe models in Sentaurus Device for the modeling of spintronics devices, in particular, for spin-selective tunneling through magnetic tunnel junctions (MTJs) and for the magnetization dynamics in ferromagnetic regions.

A simulation example for magnetization switching in an MTJ is available from the Applications Library [2].

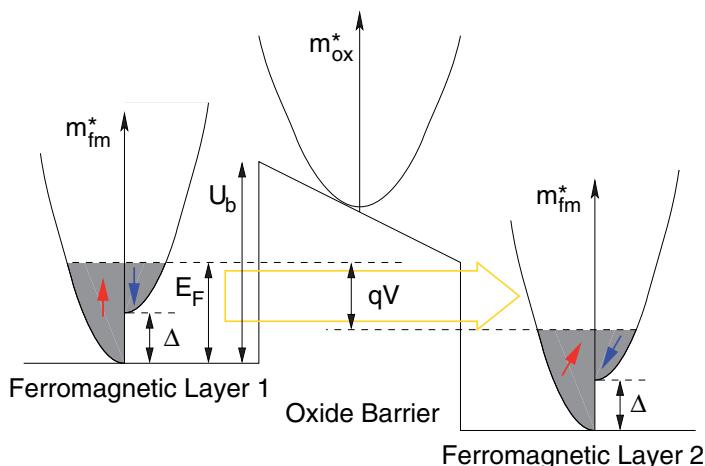
## Transport Through Magnetic Tunnel Junctions

The magnetic direct tunneling model describes the charge and spin currents flowing through a thin barrier layer sandwiched between two ferromagnetic regions. In contrast to a nonmagnetic tunnel junction, the current across an MTJ depends on both the applied voltage across the junction and the magnetization direction on either side of the barrier. This effect is called *tunneling magnetoresistance*.

### Magnetic Direct Tunneling Model

The magnetic direct tunneling model assumes the barrier layer to consist of a single material and treats the tunneling barrier as trapezoidal. [Figure 56](#) shows a schematic band diagram of such an MTJ.

*Figure 56 Schematic band diagram of an MTJ*



The current and spin transmission amplitudes for the MTJ are obtained by solving the Schrödinger equation with open boundary conditions. This can be done by applying the formalism of the non-equilibrium Green's function (NEGF) [3][4]. However, for the situation shown in [Figure 56](#), finding the numeric solution to the open boundary can be accelerated by an analytic ansatz: In the ferromagnetic regions (zero field), the spinor components of the wavefunction can be expressed as linear combinations of forward- and backward-propagating plane waves. In the barrier region (constant field), Airy functions (of the first and second kind) are used instead.

At the interfaces, continuity of probability density and conservation of probability density flux are enforced. The resulting model resembles the regular DirectTunneling model used in Sentaurus Device to study the gate leakage currents of transistors (see [Direct Tunneling on page 822](#)). However, instead of scalar wavefunctions, spinors are used to handle the spin

## Chapter 30: Ferromagnetism and Spin Transport

### Transport Through Magnetic Tunnel Junctions

degree of freedom, and the integral over the crystal momentum component parallel to the interface is retained.

Under certain restrictions ( $\Delta = 0$ ; equal masses in metal and barrier regions, moderate bias), the results of the magnetic direct tunneling model reduce to those of the regular DirectTunneling model (see [Direct Tunneling on page 822](#)). The Airy function formulation of magnetic tunneling has been validated by direct comparison to an in-house NEGF implementation. In the lattice-converged limit, there is exact agreement between the NEGF and the Airy function results.

---

## Using the Magnetic Direct Tunneling Model

The magnetic direct tunneling model is activated by specifying:

```
Tunneling(DirectTunneling(MTJ))
```

in the Physics section for the interface between the ferromagnetic and barrier materials in the command file:

```
Physics (MaterialInterface="CoFeB/MgO") {
    Tunneling(DirectTunneling(MTJ))
}
```

---

## Physics Parameters for Magnetic Direct Tunneling

The parameters for the magnetic direct tunneling model are defined in the DirectTunneling section of the parameter file pertaining to the required material (or region) interface. For example:

```
MaterialInterface="CoFeB/MgO" {
    DirectTunneling {
        m_M = 0.73
        m_dos = 0.73, 0          # hole m_dos must be zero
        m_ins = 0.16, 999        # hole value is ignored
        E_F_M = 2.25
        E_barrier = 3.285, 999   # hole value is ignored
        D_spin = 2.15
    }
}
```

The parameters for the magnetic direct tunneling model are described in [Table 158](#). Relative to the nonmagnetic direct tunneling, there is one additional parameter: the spin-energy splitting  $\Delta$ . The ferromagnetic materials are assumed to be metals; therefore, the semiconductor effective mass parameter  $m_S$  is not used, and only electron tunneling is considered. Hole parameter values are ignored, but placeholder values are required for correct parsing of the parameter file.

## Chapter 30: Ferromagnetism and Spin Transport

### Transport Through Magnetic Tunnel Junctions

#### Note:

The parameters on either side of the barrier must be the same.

The image-force effective barrier model has not been tested in the context of the magnetic direct tunneling model. It is switched off by default.

Table 158 Coefficients for direct tunneling (values for MgO on CoFeB from [4][5])

Symbol	Parameter name	Electrons	Holes	Unit	Description
$E_F$	E_F_M	2.25	-/-	eV	Fermi energy in ferromagnet (relative to conduction band)
$\Delta$	D_spin	2.15	-/-	eV	Energy splitting between spin-up and spin-down electrons in ferromagnet
$U_B$	E_barrier	3.185	<ignored>	eV	Energy barrier (difference of conduction band edge in ferromagnet and barrier)
$m_{FM}$	m_M	0.73	-/-	$m_0$	Effective mass in ferromagnet
$m_{ox}$	m_ins	0.16	<ignored>	$m_0$	Effective mass in barrier
$m_{dos}$	m_dos	0.73	0	$m_0$	Density-of-states mass parallel to the interface; the hole value <i>must</i> be zero

## Math Parameters for Magnetic Direct Tunneling

The evaluation of the MTJ tunneling integrals as well as the value caching and interpolation strategy for reusing previous results can be controlled by an `MTJ` statement in the `Math` section of the command file. For example:

```
Math {
    MTJ(interpolate(kT(order=1 Grid=1e-3)))
}
```

Table 159 summarizes the available parameters. Parameter names are subdivided by a slash (/) to reflect the hierarchical structure of the `MTJ` statement. For example, the `order` parameter in the preceding example is listed as `interpolate/kT/order`.

## Chapter 30: Ferromagnetism and Spin Transport

### Magnetization Dynamics

Table 159 Math parameters for magnetic direct tunneling model

Parameter name	Default	Unit	Description
interpolate/Voltage/Grid	1e-3	V	Grid spacing for interpolation or snapping of the voltage across the MTJ
interpolate/Voltage/order	2	–	Interpolation order for the applied voltage: <ul style="list-style-type: none"><li>• 0: Simple value snapping</li><li>• 1: Piecewise linear interpolation</li><li>• 2: Piecewise parabolic interpolation</li></ul>
interpolate/kT/Grid	1e-8	eV	Grid spacing for interpolation or snapping of the temperature (multiplied by $k_B$ ) at both ends of a tunneling edge
interpolate/kT/order	0	–	Interpolation order for the junction temperature: <ul style="list-style-type: none"><li>• 0: Simple value snapping</li><li>• 1: Bilinear interpolation</li></ul>
dE	0.01	eV	Interval size for (total) energy integration
dEp	0.01	eV	Interval size for energy integration (contribution from surface parallel k-vector)
digits	10	–	Number of significant digits in the tunneling integral

## Magnetization Dynamics

This section discusses the modeling of the magnetization dynamics inside a free ferromagnetic layer in the presence of STT. The Landau–Lifshitz–Gilbert (LLG) equation is introduced with a discussion of the expression used for the *effective magnetic field* in the macrospin approximation. It is based on the presentation of magnetization dynamics and spin-current interaction in [6][7].

## Spin Dynamics of a Free Electron in a Magnetic Field

The spin angular momentum  $\vec{S}$  and the magnetic moment  $\vec{\mu}$  of a free electron are related by the gyromagnetic ratio:

$$\gamma = \frac{|\vec{\mu}|}{|\vec{S}|} = -2.0023 \cdot \frac{e}{m_e} \cdot \frac{\hbar}{2} < 0 \quad (966)$$

$\mu_B$

The energy  $\hat{H}$  of a magnetic moment  $\vec{\mu}$  in a local field  $\vec{B} = \mu_0 \vec{H}$  is given by  $\hat{H} = -\vec{\mu} \cdot \vec{B}$ .

This Hamiltonian gives rise to the dynamic equation:

$$\frac{d\vec{S}}{dt} = \frac{i}{\hbar} [\hat{H}, \vec{S}] = -\frac{i}{\hbar} \gamma \mu_0 [\vec{S} \cdot \vec{H}, \vec{S}] = \gamma \mu_0 \vec{S} \times \vec{H} \quad (967)$$

which describes the precession of the electron spin around the direction of the local magnetic field. This can be expressed equivalently in terms of the magnetic moment instead of the spin angular momentum as:

$$\frac{d\vec{\mu}}{dt} = -\gamma_0 \vec{\mu} \times \vec{H} \quad \text{where} \quad \gamma_0 = |\gamma| \mu_0 \quad (968)$$

## Magnetization Dynamics in a Ferromagnetic Layer

The magnetization vector  $\vec{M}$  can be interpreted as a volume density  $N/V$  of magnetic dipoles  $\vec{\mu}$ . In the absence of damping terms, the magnetization vector will precess around an effective magnetic field  $\vec{H}_{\text{eff}}$  according to:

$$\frac{d\vec{M}}{dt} = -\gamma_0 \vec{M} \times \vec{H}_{\text{eff}} \quad (969)$$

where the effective field is obtained by taking the derivative of the magnetic energy density  $U$  with respect to the local magnetization:

$$\vec{H}_{\text{eff}} = -\frac{1}{\mu_0} \nabla_{\vec{M}} U \quad (970)$$

The LLG equation accounts for the observation that, over time, the magnetization aligns itself with the effective magnetic field, by adding a phenomenological *viscous damping term* [8]:

$$\frac{d\vec{M}}{dt} = -\gamma_0 \vec{M} \times \vec{H}_{\text{eff}} + \frac{\alpha}{M_s} \vec{M} \times \frac{d\vec{M}}{dt}$$

Gilbert damping

$$+ \frac{\beta}{M_s} \vec{M} \times (\vec{M} \times \vec{H}_{\text{ext}}) \quad (971)$$

## Chapter 30: Ferromagnetism and Spin Transport

### Magnetization Dynamics

where  $M_s = |\vec{M}|$  is the saturation magnetization (assumed to be a material constant), and  $\alpha$  is a phenomenological damping parameter.

In the presence of spin-polarized currents, the rate at which angular momentum is absorbed by the ferromagnetic layer also needs to be taken into account.

The spin current  $\vec{Q}$  is defined as the rate at which angular momentum is injected into the ferromagnetic layer (the magnetic direct tunneling model provides both the charge current  $I$  and the spin current  $\vec{Q}$ ).

It is assumed that the spin direction of the injected conduction electrons is aligned rapidly to the magnetization of the ferromagnetic layer. During this process, the normal component:

$$\vec{\Gamma}_s = \vec{Q} - \frac{1}{M_s^2} \vec{M}(\vec{M} \cdot \vec{Q}) = \frac{1}{M_s^2} \vec{M} \times (\vec{Q} \times \vec{M}) \quad (972)$$

of the injected angular momentum is transferred from the conduction electrons to the core electrons of the ferromagnet and exerts an additional torque on the magnetization. This is the eponymous spin transfer torque of STT-RAM:

$$\frac{d\vec{M}}{dt} = -\gamma_0 \vec{M} \times \vec{H}_{\text{eff}} + \frac{\alpha}{M_s} \vec{M} \times \frac{d\vec{M}}{dt} + \frac{\gamma_0 \vec{\Gamma}_s}{\mu_0 V} \quad (973)$$

In terms of the magnetization direction  $\vec{m} = \vec{M}/|M_s|$  and after the elimination of the time derivative on the RHS of [Equation 973](#), the LLG equation with STT takes the form:

$$(1 + \alpha^2) \frac{d\vec{m}}{dt} = -\gamma_0 \vec{m} \times \vec{H}_{\text{eff}} + \frac{\alpha \vec{\Gamma}_s}{\mu_0 M_s V} + \alpha \vec{m} \times (\vec{m} \times \vec{H}_{\text{eff}}) + \frac{\vec{\Gamma}_s}{\mu_0 M_s V} \quad (974)$$

## Contributions of the Magnetic Energy Density

As shown in [Equation 970](#), the effective magnetic field that drives the magnetization dynamics is related to the energy density  $U$ . In general, the magnetization can be a position and a time-dependent vector field  $\vec{M}(\vec{x}, t)$ . Then, the energy density  $U$  can be written as:

$$U = U_X + U_{\text{demag}} + U_{\text{aniso}} + U_{\text{ext}} \quad (975)$$

with contributions due to the following effects:

- Exchange interaction:  $U_X = A \sum_{i=x,y,z} \frac{\partial m_i}{\partial x_i}^2$

This term favors parallel alignment of nearby spins in a ferromagnet ( $A < 0$ ). If this term dominates, a single magnetic domain can span the entire sample.

## Chapter 30: Ferromagnetism and Spin Transport

### Magnetization Dynamics

- Stray/demagnetizing field:  $U_{\text{demag}} = -\frac{\mu_0}{2} \vec{M} \cdot \vec{H}_{\text{demag}}$

The demagnetizing field is the magnetic field caused by the sum of all magnetic moments in the sample. In sufficiently large samples, the energy content of the demagnetizing field can be reduced by the formation of multiple magnetic domains.

In smaller samples, this term favors aligning the magnetization direction along the longest extension of the sample (like in a *compass needle*).

- Magnetocrystalline anisotropy:  $U_{\text{aniso}}$

In crystalline materials, the magnetic energy might depend on the direction of the magnetization relative to the crystal axes.

- Zeeman energy:  $U_{\text{ext}} = -\vec{M} \cdot \vec{B}_{\text{ext}}$

Energy of the magnetic moments in an external magnetic field.

---

## Energy Density and Effective Field in Macrospin Approximation

In the macrospin approximation, each magnetic region is considered to consist of a single, perfectly aligned, magnetic domain. This removes the position dependency from the magnetization vector field  $\vec{M}(\vec{x}, t)$  of the previous section and reduces it to a single time-dependent magnetization vector  $\vec{M}(t)$ .

The macrospin approximation models the dominance of the exchange term over the remaining terms in the energy density (a single perfect domain). Since there is no more position dependency in the magnetization direction inside the sample,  $U_X$  vanishes.

In addition, the energy density  $U$  and, therefore, the effective magnetic field  $E_{\text{eff}}$  are treated as local functions of  $\vec{M}$ . This implies the assumption of a position-independent demagnetizing field inside the ferromagnetic layer. It can be shown that, in an infinite thin film as well as in an ellipsoidal [9] ferromagnetic sample, the demagnetizing field is exactly constant and parallel to the magnetization direction. For a cylindrical thin-film geometry, this is still approximately true.

At the level of the macrospin approximation, the effects of magnetocrystalline anisotropy and the demagnetizing field become indistinguishable and are grouped together into a single *effective anisotropy* term.

In Sentaurus Device, the energy density associated with this effective anisotropy is divided into the *uniaxial anisotropy*:

$$U_K = K(1 - m_z^2) \quad (976)$$

where:

$$K = \frac{1}{2}\mu_0 M_s H_k \quad (977)$$

## Chapter 30: Ferromagnetism and Spin Transport

### Magnetization Dynamics

and  $H_k$  is the Stoner–Wohlfarth switching field, and the *easy-plane anisotropy*:

$$U_P = K_P m_x^2 \quad (978)$$

with:

$$K_P = \frac{1}{2} \mu_0 M_s M_{\text{eff}} \quad (979)$$

where  $M_{\text{eff}}$  is used as a parameter to account for deviations from the ideal thin-film geometry ( $M_s = M_{\text{eff}}$ ).

---

## Using Magnetization Dynamics in Device Simulations

Magnetization dynamics is included in the simulation if the equation name `LLG` is specified in the `Solve` section of the command file. Usually, transient simulations are required for STT devices.

To solve magnetizations, the current flow in metals, and the electrostatic potential simultaneously, you must solve the LLG equation, the contact equation, and the Poisson equation as a system of coupled equations. Therefore, a typical `Solve` statement for an STT device simulation would be:

```
Transient (InitialTime=0 FinalTime=12e-9 maxstep=1.0e-11) {
    Coupled { Poisson Contact LLG }
}
```

The time-step size must be limited to ensure that the high-frequency oscillations typical of magnetization dynamics are captured.

## Domain Selection and Initial Conditions

The definition of fixed and pinned regions, and of initial conditions for the magnetization direction is handled by the `Magnetism` statement in the region-specific `Physics` sections:

```
Physics(Region="AnodeWell") {
    Magnetism(PinnedMagnetization Init(phi=0.0 theta=0.0))
}
Physics(Region="CathodeWell") {
    Magnetism(Init(phi=0.0 theta=3.14))
}
```

Here, the magnetization in the region `AnodeWell` is pinned at  $\vartheta = \phi = 0$  ( $m_z = 1$ ); whereas, the magnetization in the region `CathodeWell` is free with initial conditions of  $\phi = 0$  and  $\vartheta = 3.14$  (close to  $m_z = -1$ ).

An external magnetic field  $\vec{H}_{\text{ext}}$  (in A/m) can be specified in the `Magnetism` statement as `H_ext= (<Hx>, <Hy>, <Hz>)`.

## Chapter 30: Ferromagnetism and Spin Transport

### Magnetization Dynamics

## Plotting Time-Dependent Magnetization

In the macrospin approximation, the magnetization is constant across each region. Then, the full time evolution of the magnetization can be captured by plotting the average direction of the magnetization vector in the free layer. In the simulation example [2], the free layer is called `CathodeWell`, and plotting of the magnetization direction is triggered by the following `CurrentPlot` section:

```
CurrentPlot { MagnetizationDir/Vector3D(average(Region="CathodeWell")) }
```

In addition to the magnetization direction in Cartesian coordinates, you can plot the magnetization of the zenith angle `Magnetization_theta` and the azimuth `Magnetization_phi` of the magnetization direction ( $\vartheta = \phi = 0$  corresponds to the positive  $z$ -direction).

---

## Parameters for Magnetization Dynamics

The parameters for magnetization dynamics (LLG equation) are defined in the `Magnetism` section of the corresponding material or region in the parameter file.

Table 160 Parameters for LLG equation (typical values shown)

Symbol	Parameter name	Value and unit	Description
$M_s$	SaturationMagnetization	8.0e5 A/m	Saturation magnetization of the ferromagnetic material
$\alpha$	alpha	0.01	Gilbert damping coefficient
$H_K$	Hk	7957.75 A/m	Stoner–Wohlfarth switching field (depends on layer geometry)
$M_{\text{eff}}$	Meff	5e5 A/m	Effective magnetization for parameterizing the easy-plane anisotropy (geometry dependent)
$A$	A	1e-11 J/m	Exchange stiffness of the ferromagnetic material (only used if macrospin is deactivated; see <a href="#">Magnetization Dynamics Beyond Macrospin: Position-Dependent Exchange and Spin Waves</a> on page 932)

## Time-Step Control for Magnetization Dynamics

To improve time-step control during the transient solution of the LLG equation, you can restrict the maximum change of the magnetization direction during the simulation that might occur during a single time step. If this limit is exceeded, the update is rejected, and the time step  $\Delta t$  is reduced until the limit is met.

For example, a maximum local change of the Cartesian components of the magnetization of 0.1 per time step is requested in the `Math` section of the command file like this:

```
Math {
    Magnetism(dxyz=0.1)
}
```

The default value for `dxyz` is 0.15.

## Thermal Fluctuations

The magnetization dynamics can be influenced by thermal fluctuations. This effect can be included in the analysis by replacing the deterministic effective field  $\vec{H}_{\text{eff}}$  with  $\vec{H}_{\text{eff}} + \vec{H}_{\text{T}}$ , where the *thermal fluctuation field*  $\vec{H}_{\text{T}}$  is a stochastic field with the autocorrelation function [10]:

$$H_T^i(t)H_T^j(t') = \frac{2kT\alpha}{\gamma_0\mu_0 M_s V} \delta_{i,j} \delta(t-t') \quad (980)$$

## Using Thermal Fluctuations

Modeling of thermal fluctuations is activated by adding the `ThermalFluctuations` keyword to the `Magnetism` statement of the `Physics` section:

```
Physics {
    Magnetism(ThermalFluctuations)
}
```

The magnitude of the thermal fluctuation field can be modified by multiplication with the optional parameter `H_th_scaling_factor` (default: 1). The syntax for suppressing  $\vec{H}_{\text{T}}$  by a factor of 2 (which corresponds to dividing the temperature by 4) is:

```
Physics {
    Magnetism(ThermalFluctuations H_th_scaling_factor=0.5)
}
```

By default, the seed for the random number generator used in the randomization process differs for every simulation. However, you can specify `Math/Magnetism/RandomSeed` if required. This allows a particular randomization to be repeated if the same seed is used in a subsequent simulation of the same device on the same computer.

## Chapter 30: Ferromagnetism and Spin Transport

### Parallel and Perpendicular Spin Transfer Torque

---

In an MTJ, it is customary to decompose the STT  $\vec{\Gamma}_s$  (Equation 972) into *perpendicular* and *parallel* (or *in-plane*) components, relative to the plane spanned by the magnetization directions on both sides of the tunneling barrier.

If  $\vec{n} = \vec{m}_1 \times \vec{m}_2$  is the unit normal vector of this plane, then the perpendicular torque is defined as:

$$\vec{\Gamma}_{\perp} = \vec{n} \cdot \vec{\Gamma}_s \cdot \vec{n} \quad (981)$$

The in-plane torque is defined as:

$$\vec{\Gamma}_{\parallel} = -\vec{n} \times \vec{n} \times \vec{\Gamma}_s = \vec{\Gamma}_s - \vec{\Gamma}_{\perp} \bullet \vec{n} \quad (982)$$

Sometimes, it might be instructive to be able to modify the relative strength of the in-plane and the perpendicular torque components. For this purpose, user-accessible scaling factors  $\varepsilon_{\parallel}$  and  $\varepsilon_{\perp}$  are provided. If they are modified from their default values of 1, the STT  $\vec{\Gamma}_s$  in the LLG equation (Equation 974) is replaced with an effective torque:

$$\vec{\Gamma}_{s, \text{eff}} = \varepsilon_{\parallel} \vec{\Gamma}_{\parallel} + \varepsilon_{\perp} \vec{\Gamma}_{\perp} \quad (983)$$

In the command file, these scaling factors can be accessed from the `Physics` section:

```
Physics {
    Magnetism(parallel_torque_scaling_factor=<double>)           # ε||
    Magnetism(perpendicular_torque_scaling_factor=<double>)        # ε⊥
}
```

---

## Magnetization Dynamics Beyond Macrospin: Position-Dependent Exchange and Spin Waves

The macrospin approximation is based on the assumption that the effect of the exchange interaction is much stronger than all other magnetic effects. Therefore, the most energetically favorable magnetization configuration is a single perfectly aligned domain that spans the entire ferromagnet.

In many structures, however, there is competition between the exchange interaction, which favors single-domain behavior, and the demagnetizing field, which tries to break down domains to reduce the energy content of the stray field. The nonlocality of the demagnetizing field has not yet been implemented in Sentaurus Device.

## Chapter 30: Ferromagnetism and Spin Transport

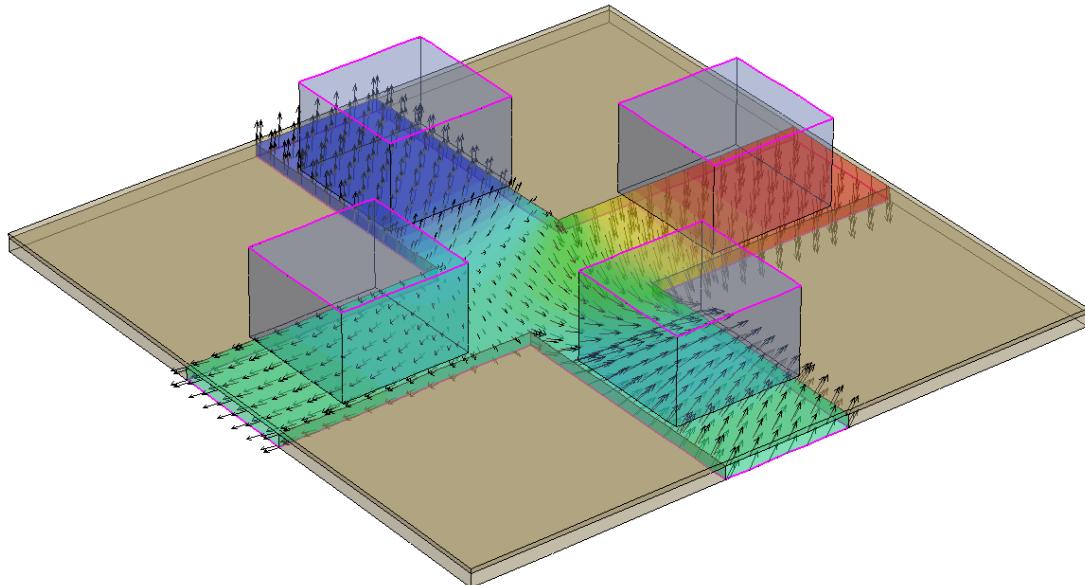
Magnetization Dynamics Beyond Macrospin: Position-Dependent Exchange and Spin Waves

However, even without it, restoring the vector-field character to the magnetization direction  $\vec{m}(x, t)$  and adding the exchange field:

$$\vec{H}_X = \frac{2A}{\mu_0 M_s} \nabla^2 \vec{m} \quad (984)$$

to the effective field  $\vec{H}_{\text{eff}}$  of the LLG equation (Equation 974) leads to interesting phenomena such as spin waves. This allows, for example, the modeling of the nonlocal switching behavior of devices such as the spin-torque majority gate suggested in [11].

Figure 57 Snapshot of position-dependent magnetization direction in a spin-torque majority gate; the pinned layer is split into four pieces, and the MTJs with different applied voltages compete for switching of the cross-shaped free layer



---

## Using Position-Dependent Exchange

The exchange term becomes active as soon as the node-merging mechanism of the macrospin approximation is deactivated:

```
Physics (Region="CathodeWell") {  
    Magnetism(-MacroSpin)      # deactivate macrospin approximation  
}
```

## User-Defined Contributions to Effective Magnetic Field of LLG Equation

You can define additional contributions to the  $\vec{H}_{\text{eff}}$  field of the LLG equation ([Equation 974](#)) by using the physical model interface (PMI). See [Chapter 39 on page 1207](#) for general information about the PMI and [Ferromagnetism and Spin Transport on page 1417](#) for details about spintronics-specific PMI models.

---

## References

- [1] D. C. Ralph and M. D. Stiles, "Spin transfer torques," *Journal of Magnetism and Magnetic Materials*, vol. 320, no. 7, pp. 1190–1216, 2008.
- [2] *Simulation of Magnetization Switching in a CoFeB/MgO/CoFeB Magnetic Tunnel Junction*, available from TCAD Sentaurus Version T-2022.03 installation, go to Applications\_Library/Memory/STT\_MTJ.
- [3] D. Datta *et al.*, "Quantitative Model for TMR and Spin-transfer Torque in MTJ devices," in *IEDM Technical Digest*, San Francisco, CA, USA, pp. 548–551, December 2010.
- [4] Y. Hiramatsu *et al.*, "NEGF Simulation of Spin-Transfer Torque in Magnetic Tunnel Junctions," in *International Meeting for Future of Electron Devices*, Osaka, Japan, pp. 102–103, May 2011.
- [5] D. Datta *et al.*, "Voltage Asymmetry of Spin-Transfer Torques," *IEEE Transactions on Nanotechnology*, vol. 11, no. 2, pp. 261–272, 2012.
- [6] J. Z. Sun, "Spin-current interaction with a monodomain magnetic body: A model study," *Physical Review B*, vol. 62, no. 1, pp. 570–578, 2000.
- [7] J. Miltat, G. Albuquerque, and A. Thiaville, "An Introduction to Micromagnetics in the Dynamic Regime," *Spin Dynamics in Confined Magnetic Structures I*, vol. 83, B. Hillebrands and K. Ounadjela (eds.), Springer: Berlin, pp. 1–34, 2002.
- [8] T. L. Gilbert, "A Phenomenological Theory of Damping in Ferromagnetic Materials," *IEEE Transactions on Magnetics*, vol. 40, no. 6, pp. 3443–3449, 2004.
- [9] J. A. Osborn, "Demagnetizing Factors of the General Ellipsoid," *Physical Review*, vol. 67, no. 11 and 12, pp. 351–357, 1945.
- [10] J. Xiao, A. Zangwill, and M. D. Stiles, "Macrospin models of spin transfer dynamics," *Physical Review B*, vol. 72, no. 1, p. 014446, 2005.
- [11] D. E. Nikonov, G. I. Bourianoff, and T. Ghani, "Proposal of a Spin Torque Majority Gate Logic," *IEEE Electron Device Letters*, vol. 32, no. 8, pp. 1128–1130, 2011.

# 31

## Mechanical Stress

---

*This chapter presents an overview of the importance of stress in device simulation.*

---

### Overview of Mechanical Stress

Stress engineering is a key point to ensuring the high performance of CMOS devices. Mechanical stress can affect workfunction, band gap, effective mass, carrier mobility, and leakage currents. The stress is generated by many technological processes due to different process temperatures and material properties. In addition, it can be added (such as silicon layers onto SiGe bulk) to improve device performance.

Mechanical distortion of semiconductor microstructures results in a change in the band structure and carrier mobility. These effects are well known, and appropriate computations of the change in the strain-induced band structure are based on the deformation potential theory [1]. The implementation of the deformation potential model in Sentaurus Device is based on data and approaches presented in the literature [1][2][3][4]. Other approaches [5][6] implemented in Sentaurus Device focus more on the description of piezoresistive effects.

---

### Stress and Strain in Semiconductors

Generally, the stress tensor  $\bar{\sigma}$  is a symmetric  $3 \times 3$  matrix. Therefore, it only has six independent components, and it is convenient to express it in a contracted six-component vector notation:

$$\bar{\sigma} = \begin{bmatrix} \sigma_{xx} & \sigma_{xy} & \sigma_{xz} \\ \sigma_{xy} & \sigma_{yy} & \sigma_{yz} \\ \sigma_{xz} & \sigma_{yz} & \sigma_{zz} \end{bmatrix} \rightarrow \begin{bmatrix} \sigma_{xx} \\ \sigma_{yy} \\ \sigma_{zz} \\ \sigma_{yz} \\ \sigma_{xz} \\ \sigma_{xy} \end{bmatrix} = \begin{bmatrix} \sigma_{11} \\ \sigma_{22} \\ \sigma_{33} \\ \sigma_{23} \\ \sigma_{13} \\ \sigma_{12} \end{bmatrix} = \begin{bmatrix} \sigma_1 \\ \sigma_2 \\ \sigma_3 \\ \sigma_4 \\ \sigma_5 \\ \sigma_6 \end{bmatrix} \quad (985)$$

where pairs of indices are contracted using:

$$11 \rightarrow 1, 22 \rightarrow 2, 33 \rightarrow 3, 23 \rightarrow 4, 13 \rightarrow 5, 12 \rightarrow 6 \quad (986)$$

The contracted tensor notation simplifies tensor expressions. For example, one of the options for computing the strain tensor  $\bar{\varepsilon}$  (which is needed for the deformation potential model) is given by the generalized Hooke's law for anisotropic materials:

$$\varepsilon_{ij} = \sum_{k=1}^3 S_{ijkl} \sigma_{kl} \quad (987)$$

where  $S_{ijkl}$  is a component of the elastic compliance tensor  $\bar{S}$ . The elastic compliance tensor is symmetric, which allows [Equation 987](#) to be written in a simplified contracted form:

$$\varepsilon_i = \sum_{j=1}^6 S_{ij} \sigma_j \quad (988)$$

In addition to index contraction ([Equation 986](#)), the following contraction rules for  $\varepsilon_{ij}$  and  $S_{ijkl}$  were used to obtain this result [7]:

$$\begin{aligned} \varepsilon_p &= \varepsilon_{ij} && \text{, if } p < 4 \\ &= 2\varepsilon_{ij} && \text{, if } p > 3 \\ S_{pq} &= S_{ijkl} && \text{, if } p < 4 \text{ and } q < 4 \\ &= 4S_{ijkl} && \text{, if } p > 3 \text{ and } q > 3 \\ &= 2S_{ijkl} && \text{, otherwise} \end{aligned} \quad (989)$$

Note that  $\varepsilon_4$ ,  $\varepsilon_5$ , and  $\varepsilon_6$  are often called *engineering shear-strain components* and are related to the double-subscripted shear components that are used in the model equations by:

$$\begin{aligned} \varepsilon_4 &= 2\varepsilon_{23} = 2\varepsilon_{32} \\ \varepsilon_5 &= 2\varepsilon_{13} = 2\varepsilon_{31} \\ \varepsilon_6 &= 2\varepsilon_{12} = 2\varepsilon_{21} \end{aligned} \quad (990)$$

In crystals with cubic symmetry such as silicon, the number of independent coefficients of the elastic compliance tensor (as well as with other material property tensors) reduces to three by rotating the coordinate system parallel to the high-symmetric axes of the crystal [8].

This gives the following (contracted) compliance tensor  $\bar{S}$ :

$$\bar{S} = \begin{bmatrix} S_{11} & S_{12} & S_{12} & 0 & 0 & 0 \\ S_{12} & S_{11} & S_{12} & 0 & 0 & 0 \\ S_{12} & S_{12} & S_{11} & 0 & 0 & 0 \\ 0 & 0 & 0 & S_{44} & 0 & 0 \\ 0 & 0 & 0 & 0 & S_{44} & 0 \\ 0 & 0 & 0 & 0 & 0 & S_{44} \end{bmatrix} \quad (991)$$

where the coefficients  $S_{11}$ ,  $S_{12}$ , and  $S_{44}$  correspond to parallel, perpendicular, and shear components, respectively.

In Sentaurus Device, the stress tensor can be defined in the stress coordinate system  $(\vec{e}_1, \vec{e}_2, \vec{e}_3)$ . To transfer this tensor to another coordinate system (for example, the crystal system  $(\vec{e}'_1, \vec{e}'_2, \vec{e}'_3)$ , which is a common operation), the following transformation rule between two coordinate systems is applied:

$$\overset{3}{\underset{k=1}{\overset{l=1}{\underset{j=1}{\overset{i=1}{\sigma'_{ij}}}}} = \overset{3}{\underset{k=1}{\overset{l=1}{\underset{j=1}{\overset{i=1}{a_{ik}a_{jl}\sigma_{kl}}}}}} \quad (992)$$

where  $\bar{a}$  is the rotation matrix:

$$a_{ik} = \frac{\vec{e}'_i \cdot \vec{e}_k}{\|\vec{e}'_i\| \|\vec{e}_k\|} \quad (993)$$

## Using Stress and Strain

Stress-dependent models are selected in the `Physics{Piezo()}` section of the command file. Components of the stress and strain tensors (if they are constant over the device or region) also are specified here, as well as the components  $\vec{e}_1$  `OriKddX` and  $\vec{e}_2$  `OriKddY` of the coordinate system where stress and strain are defined (see [Table 161](#)):

```
Physics {
    Piezo (
        Stress = (XX, YY, ZZ, YZ, XZ, XY)
        Strain = (XX, YY, ZZ, YZ, XZ, XY)
        OriKddX = (1,0,0)
        OriKddY = (0,1,0)
        Model (...)
    )
}
```

*Table 161 General keywords for Piezo*

Parameter	Description
Stress=(XX, YY, ZZ, YZ, XZ, XY)	Specifies uniform stress [Pa] if the <code>Piezo</code> file is not given in the <code>File</code> section.
Strain=(XX, YY, ZZ, YZ, XZ, XY)	Specifies uniform strain <a href="#">[1]</a> .
OriKddX = (1,0,0)	Defines Miller indices of the stress system relative to the simulation system.
OriKddY = (0,1,0)	Defines Miller indices of the stress system relative to the simulation system.
Model(<options>)	Selects stress-dependent models in <options> (see sections from <a href="#">Deformation of Band Structure on page 940</a> to <a href="#">Mobility Modeling on page 952</a> ).

**Note:**

The stress system is always defined relative to the simulation coordinate system of Sentaurus Device (in the `Piezo` section of the command file). The simulation coordinate system is defined relative to the crystal orientation system by default but, in the parameter file (see below), it is possible to define the crystal system relative to the simulation system. By default, all three coordinate systems coincide.

## Stress Tensor

Apart from specifying a constant stress tensor in the `Piezo` section of the command file, Sentaurus Device also provides different ways to define position-dependent stress values:

- A field of stress values [Pa] (as obtained by mechanical structure analysis) is read by specifying the `Piezo` entry in the `File` section:

```
File {
    Piezo = <piezofile>
}
```

Sentaurus Device recognizes stress either as a single symmetric second-order tensor of dimension 3 (`Stress`), or as six scalar values (`StressXX`, `StressXY`, `StressXZ`, `StressYY`, `StressYZ`, and `StressZZ`).

- A physical model interface can be used for stress specification (see [Stress on page 1388](#)).

- The `Mechanics` command in the `Solve` section can update the stress tensor in response to changes in bias conditions. In particular, the stress tensor can be computed as a function of lattice temperature and electric field. This is described in [Mechanics Solver on page 1008](#).

**Note:**

The stress values in all these stress specifications should be in Pa (1 Pa = 10 dyn/cm<sup>2</sup>) and tensile stress should be positive according to convention.

## Strain Tensor

The strain tensor can be computed in one of the following ways:

- According to the generalized Hooke's law, strain can be obtained from stress through the elastic compliance tensor  $S$  (see [Equation 987 on page 936](#)).
- A constant strain tensor can be specified in the `Piezo` section of the command file (see [Table 161 on page 938](#)).
- Sentaurus Device can read the strain tensor  $\bar{\epsilon}$  from the TDR file (`ElasticStrain` field). This requires that a `Piezo` file is specified in the `File` section.

By default, the strain tensor will be computed by Hooke's law. If necessary, the required option can be selected as follows in the `Piezo` section of the command file:

```
Physics {
    Piezo (
        Strain = Hooke
        Strain = (XX, YY, ZZ, YZ, XZ, XY)
        Strain = LoadFromFile
    )
}
```

## Stress Limits

Extremely high stress values can sometimes cause stress-dependent models to produce nonphysical results. As an option, you can limit stress values read from files or specified in the command file to a user-specified maximum. This is accomplished by specifying the `StressLimit` parameter in the `Math` section of the command file:

```
Math {
    StressLimit = 4e9    #[Pa]
}
```

The magnitude of all stress components at each semiconductor vertex is limited to the specified value, but the sign of the stress value is retained. For example, with the above specification, a stress value of  $\sigma_{yy} = -8.2 \times 10^9$  read from a file is limited to  $\sigma_{yy} = -4 \times 10^9$ .

## Crystallographic Orientation and Compliance Coefficients

The simulation coordinate system relative to the crystal coordinate system can be defined by the `x` and `y` vectors in the `LatticeParameters` section of the parameter file. The defaults are:

```
LatticeParameters {  
    x = (1, 0, 0)  
    y = (0, 1, 0)  
}
```

The simulation system is defined relative to the crystal system. Alternatively, there is an option to represent the crystal system relative to the Sentaurus Device simulation system. In this case, the keyword `CrystalAxis` must be in the `LatticeParameters` section, and the `x` and `y` vectors will represent the `100` and `010` axes of the crystal system in the Sentaurus Device simulation system.

The elastic compliance coefficients  $S_{ij}$  [ $10^{-12} \text{ cm}^2/\text{dyn}$ ] can be specified in the field `s[i][j]` in the `LatticeParameters` section of the parameter file. If you select the cubic crystal system (by specifying `CrystalSystem=0`), then it is sufficient to specify  $S_{11}$ ,  $S_{12}$ , and  $S_{44}$ . For a hexagonal crystal system (`CrystalSystem=1`),  $S_{33}$  and  $S_{13}$  must also be specified. Otherwise, all unspecified coefficients are set to 0.

The following section of the parameter file shows the defaults for silicon:

```
LatticeParameters {  
    * Crystal system and elasticity.  
    X = (1, 0, 0) # [1]  
    Y = (0, 1, 0) # [1]  
    S[1][1] = 0.77 # [1e-12 cm^2/dyn]  
    S[1][2] = -0.21 # [1e-12 cm^2/dyn]  
    S[4][4] = 1.25 # [1e-12 cm^2/dyn]  
    CrystalSystem = 0 # [1]  
}
```

---

## Deformation of Band Structure

In deformation potential theory [2][9], the strains are considered to be relatively small. The change in energy of each carrier valley or band, caused by the small deformation of the lattice, is a linear function of the strain.

## Chapter 31: Mechanical Stress

### Deformation of Band Structure

For silicon, Bir and Pikus [9] proposed a model for the strain-induced change in the energy of carrier valleys or bands (three  $\Delta_2$  electron valleys, heavy-hole, and light-hole bands are considered) where they ignore the shear strain for electrons and suggest nonlinear dependence for holes (which corresponds to  $6 \times 6$  k·p theory [12]):

$$\begin{aligned}\Delta E_{C,i} &= \Xi_d(\varepsilon'_{11} + \varepsilon'_{22} + \varepsilon'_{33}) + \Xi_u \varepsilon'_{ii} \\ \Delta E_{V,i} &= a(\varepsilon'_{11} + \varepsilon'_{22} + \varepsilon'_{33}) \pm \delta E \\ \delta E &= \sqrt{\frac{b^2}{2}((\varepsilon'_{11} - \varepsilon'_{22})^2 + (\varepsilon'_{22} - \varepsilon'_{33})^2 + (\varepsilon'_{11} - \varepsilon'_{33})^2) + d^2(\varepsilon'_{12}^2 + \varepsilon'_{13}^2 + \varepsilon'_{23}^2)}\end{aligned}\quad (994)$$

where  $\Xi_d$ ,  $\Xi_u$ ,  $a$ ,  $b$ ,  $d$  are deformation potentials,  $i$  corresponds to the carrier band number, and  $\varepsilon_{ij}$  are the components of the strain tensor in the crystal coordinate system (see [Stress and Strain in Semiconductors on page 935](#) for a description of tensor transformations). The sign  $\pm$  separates heavy-hole and light-hole bands of silicon.

The stress-induced change of the  $\Delta_2$  electron valley energy in [Equation 994](#) corresponds to a simple form of the linear deformation model. The model applied to arbitrary ellipsoidal bands (for example, as for four L-electron valleys in germanium or  $\Gamma$ -electron valley in III–V materials) could be expressed as [10]:

$$\Delta E_i = \left[ \Xi_d \bar{I} + \Xi_u \overset{\rightarrow}{e}_i \overset{\rightarrow}{e}_i^T \right] : \bar{\varepsilon}' \quad (995)$$

where:

- $\Xi_d$ ,  $\Xi_u$  are linear deformation potentials.
- $\bar{I}$  is a unit  $3 \times 3$  matrix.
- $\bar{\varepsilon}'$  is the strain tensor in the crystal coordinate system.
- $\overset{\rightarrow}{e}_i$  is the unit vector parallel to the  $\mathbf{k}$ -vector of the main axis of the ellipsoidal valley  $i$ .

#### Note:

The dyadic product is defined as  $\bar{a} : \bar{b} = \sum_j a_{ij} b_{ij}$ .

For spherical bands such as, for example, the  $\Gamma$ -electron valley with isotropic effective mass, the deformation potential  $\Xi_u$  should be equal to zero in [Equation 995](#).

As previously mentioned, the linear deformation potential model ([Equation 995](#)) is limited to small strain and some specific band structures. For silicon, other models provide nonlinear corrections.

## Chapter 31: Mechanical Stress

### Deformation of Band Structure

Using a degenerate  $k \cdot p$  theory at the zone boundary X-point, the authors of [11] and [13] derived an additional shear term for the  $\Delta_2$  electron valleys in [Equation 994](#):

$$\Delta E_{C,i} = \Xi_d(\epsilon'_{11} + \epsilon'_{22} + \epsilon'_{33}) + \Xi_u \epsilon'_{ii} + \begin{cases} \circ & -\frac{\Delta}{4} \eta_i^2, |\eta_i| \leq 1 \\ \circ & -(2|\eta_i| - 1) \frac{\Delta}{4}, |\eta_i| > 1 \end{cases} \quad (996)$$

where:

- $\eta_i = \frac{4\Xi_u \epsilon'_{jk}}{\Delta}$  is a dimensionless off-diagonal strain with  $j \neq k \neq i$ .
- $\epsilon'_{jk}$  is a shear strain component.
- $\Delta$  is the band separation between the two lowest conduction bands.
- $\Xi_u'$  is the deformation potential responsible for the band-splitting of the two lowest conduction bands [11]:  $(E_{\Delta_1} - E_{\Delta_2})|_{X[001]} = 4\Xi_u' \epsilon'_{jk}$ .

The strain-induced shifts of valence bands can be computed using  $6 \times 6$   $k \cdot p$  theory for the heavy-hole, light-hole, and split-off bands as described in [\[12\]](#).

The specification of the deformation potentials for these models and is described in [Using Deformation Potential Model on page 943](#).

Using the stress tensor  $\bar{\sigma}$  from the command file, Sentaurus Device recomputes it from the stress coordinate system to the tensor  $\bar{\sigma}'$  in the crystal system by [Equation 992](#). The strain tensor  $\bar{\epsilon}'$  is a result of applying Hooke's law [Equation 988](#) to the stress  $\bar{\sigma}'$ . Using [Equation 995](#), [Equation 996](#), or solving the cubic equation from [\[12\]](#), the energy band change can be computed for each conduction and valence carrier bands.

By default, Sentaurus Device does not modify the effective masses, but instead it computes strain-induced conduction and valence band-edge shifts,  $\Delta E_C$  and  $\Delta E_V$ , using an averaged value of the individual band-edge shifts:

$$\frac{\Delta E_C}{kT_{300}} = -\ln \left[ \frac{1}{n_C} \prod_{i=1}^{n_C} \exp \left( \frac{-\Delta E_{C,i}}{kT_{300}} \right) \right] \quad (997)$$

$$\frac{\Delta E_V}{kT_{300}} = \ln \left[ \frac{1}{n_V} \prod_{i=1}^{n_V} \exp \left( \frac{\Delta E_{V,i}}{kT_{300}} \right) \right]$$

where  $n_C$  and  $n_V$  are the number of subvalleys considered in the conduction and valence bands, respectively, and  $T_{300} = 300$  K.

## Chapter 31: Mechanical Stress

### Deformation of Band Structure

Alternatively, a more accurate representation of the band gap can be obtained by using the minimum and maximum of the individual conduction and valence band shifts, respectively, as follows:

$$\begin{aligned}\Delta E_C &= \min(\Delta E_{C,i}) \\ \Delta E_V &= \max(\Delta E_{V,i})\end{aligned}\quad (998)$$

In this case, however, the strain dependency of the effective mass and the density-of-states should be accounted for (see [Strained Effective Masses and Density-of-States on page 945](#)).

The band gap and affinity can be modified:

$$\begin{aligned}E_g &= E_{g0} + \Delta E_C - \Delta E_V \\ \chi &= \chi_0 - \Delta E_C\end{aligned}\quad (999)$$

where the index '0' corresponds to the affinity and bandgap values before stress deformation.

---

## Using Deformation Potential Model

To activate the deformation potential models [Equation 994–Equation 996](#) with k·p models for electrons and holes, and [Equation 998](#) for the conduction band and valence band energy shifts, the following must be specified in the `Piezo` section of the command file:

```
Physics (Region = "StrainedSilicon") {
    Piezo(
        Model(DeformationPotential(ekp hkp minimum)
    )
}
```

### Note:

Usage of `DeformationPotential` without the `ekp` and `hkp` options is not recommended because an unsupported way of setting deformation potentials in [Equation 994](#) will be used.

To modify the deformation potentials of these models, the following parameters in the section `LatticeParameters` should be used:

```
LatticeParameters {
    * Deformation potentials of k.p model for electron bands
    xis = 7      # [eV]
    dbs = 0.53   # [eV]
    xiu = 9.16   # [eV]
    xid = 0.77   # [eV]
    Mkp = 1.2    # [1]
    * Deformation potentials of k.p model for hole bands
    adp = 2.1    # [eV]
    bdp = -2.33  # [eV]
```

## Chapter 31: Mechanical Stress

### Deformation of Band Structure

```
ddp = -4.75 # [eV]
dso = 0.044 # [eV]
* Deformation potentials and energy (in ref. to Delta-valley) for
* L-valleys
xiu_l = 11.5 # [eV]
xid_l = -6.58 # [eV]
e_l = 1.1 # [eV]
* Deformation potential and energy (in ref. to Delta-valley) for
* Gamma-valley
xid_gamma = -7.0 # [eV]
e_gamma = 2.3 # [eV]
}
```

This example shows the default parameter values for silicon. The parameters `xis`, `dbs`, `xiu`, and `xid` correspond to the deformation potentials  $\Xi_{u'}$ ,  $\Delta$ ,  $\Xi_u$ ,  $\Xi_d$  of the  $\Delta_2$  electron valleys in [Equation 996](#). The parameters `xiu_l` and `xid_l` define the deformation potentials of the L-valleys in [Equation 995](#), and `e_l` sets the relaxed energy difference between the L and  $\Delta_2$  electron valleys in the conduction band.

The parameter `xid_gamma` defines the deformation potential of the  $\Gamma$ -valley, and the parameter `e_gamma` sets the relaxed energy difference between the  $\Gamma$  and  $\Delta_2$  electron valleys. The other parameters are the valence-band deformation potentials described in [\[12\]](#):

- `adp` is the hydrostatic deformation potential.
- `bdp` is the shear deformation potential.
- `ddp` is the deformation potential.
- `dso` is the spin-orbit splitting energy.

Using  $\Delta_2$ -valleys, L-valleys, and  $\Gamma$ -valley for the conduction band representation, and  $6 \times 6$   $k \cdot p$  hole bands for the valence band, you can describe various semiconductor band structures. Sentaurus Device provides default band-structure parameters for silicon, germanium, SiGe, and several III-V materials in the `LatticeParameters` section. All parameters in this section can be mole fraction dependent.

In applications where the multivalley model is used (see [Multivalley Band Structure on page 950](#)), there is a possibility to use the option `DeformationPotential(multivalley)` as an alternative to the other options. This option checks whether the multivalley model is activated in the `Physics` section (if not, the `multivalley` option will be ignored) and, if so, uses all the valleys specified in the model (including  $k \cdot p$  and analytic ones with the deformation potentials) to define the change to the band edge. This option allows you to propagate the Multivalley bandgap and Affinity change to other models, such as generation–recombination models.

## Chapter 31: Mechanical Stress

### Deformation of Band Structure

#### Note:

Specifying DeformationPotential(minimum ekp hkp multivalley) with, for example, the eMultivalley model activated will overwrite the options (minimum ekp) for electrons and will use only the valleys of the eMultivalley specification to define the change to the conduction band edge (accounting for the minimum energy of each valley). The same specification without, for example, the hMultivalley model means that the options (minimum hkp) will be used to define the change to the valence band edge.

To see the changes to the conduction band and valence band edges related to the DeformationPotential model separately, specify the following in the Plot section:

```
Plot {  
    eDeformationPotential hDeformationPotential  
}
```

---

## Strained Effective Masses and Density-of-States

Sentaurus Device provides options for computing strain-dependent effective mass for both electrons and holes and, consequently, the strain-dependent conduction band and valence band effective density-of-states (DOS).

## Strained Electron Effective Mass and DOS

The conduction band in silicon is approximated by three pairs of equivalent  $\Delta_2$  valleys. Without stress, the DOS of each valley is:

$$N_{C,i} = \frac{N_C}{3}, \quad i = 1, 2, 3 \quad (1000)$$

where  $N_C$  can be defined by two effective mass components  $m_l$  and  $m_t$  (see [Equation 177 on page 320](#)).

As described in [Deformation of Band Structure on page 940](#), an applied stress induces a relative shift of the energy  $\Delta E_{C,i}$  that is different for each  $\Delta_2$  valley. In addition, in the presence of shear stress, there is a large effective mass change for electrons. An analytic derivation for this mass change can be found in [13] and is based on a two-band k·p theory. To simplify the final expressions, the same dimensionless off-diagonal strain introduced in [Equation 996](#) is used here:

$$\eta_i = \frac{4\Xi_u \epsilon'_{jk}}{\Delta}, \quad j \neq k \neq i \quad (1001)$$

Note that the  $\epsilon'_{jk}$  strain affects only the  $\Delta_2$  valley along the  $i$ -axis, where  $i = 1, 2$ , or  $3$  represents the [100], [010], or [001] axis, respectively.

## Chapter 31: Mechanical Stress

### Deformation of Band Structure

When the  $\mathbf{k} \cdot \mathbf{p}$  model [13] is evaluated at the band minima of the first conduction band, two different branches for the transverse effective mass are obtained where  $m_{t1,i}$  is the mass across the stress direction, and  $m_{t2,i}$  is the mass along the stress direction:

$$m_{t1,i}/m_t = \begin{cases} \frac{\circ}{\circ} & 1 - \frac{\eta_i}{M}^{-1}, |\eta_i| \leq 1 \\ \rightarrow & \\ \circ & 1 - \frac{\text{sign}(\eta_i)}{M}^{-1}, |\eta_i| > 1 \end{cases} \quad (1002)$$

$$m_{t2,i}/m_t = \begin{cases} \circ & 1 + \frac{\eta_i}{M}^{-1}, |\eta_i| \leq 1 \\ \circ & 1 + \frac{\text{sign}(\eta_i)}{M}^{-1}, |\eta_i| > 1 \end{cases} \quad (1003)$$

$$m_{l,i}/m_l = \begin{cases} \circ & (1 - \eta_i^2)^{-1}, |\eta_i| < 1 \\ \circ & 1 - \frac{1}{|\eta_i|}^{-1}, |\eta_i| > 1 \end{cases} \quad (1004)$$

In the above equations,  $M$  is a  $\mathbf{k} \cdot \mathbf{p}$  model parameter that has been adjusted to provide a good fit with the empirical pseudopotential method (EPM) results.

Another result of the  $\mathbf{k} \cdot \mathbf{p}$  model [13] is that the nonparabolicity of the  $\Delta_2$  valley changes with stress as follows:

$$\alpha = \alpha_0 \frac{1 + 2(\eta m_t/M)^2}{1 - (\eta m_t/M)^2} \quad (1005)$$

where  $\alpha_0$  is the relaxed nonparabolicity of the  $\Delta_2$  valley. Such stress-induced change of the nonparabolicity in Equation 1005 can be accounted for with the `eMultivalley(kpDOS Nonparabolicity)` option (see [Multivalley Band Structure on page 327](#)).

The stress-induced change of the effective DOS for each  $\Delta_2$ -valley can then be written as:

$$N_{C,i} = \sqrt{\frac{m_{t1,i}}{m_t} \frac{m_{t2,i}}{m_t} \frac{m_{l,i}}{m_l}} \cdot \frac{N_C}{3} \quad (1006)$$

Accounting for the change of the stress-induced valley energy  $\Delta E_{C,i}$  and the carrier redistribution between valleys, the strain-dependent conduction-band effective DOS can be derived for Boltzmann statistics:

$$N'_C = \gamma_n \cdot N_C \quad (1007)$$

## Chapter 31: Mechanical Stress

### Deformation of Band Structure

where:

$$\gamma_n = \frac{1}{N_C} \cdot \sum_{i=1}^3 N_{C,i} \cdot \exp\left(\frac{\Delta E_{C,\min} - \Delta E_{C,i}}{kT_n}\right) \quad (1008)$$

and:

$$\Delta E_{C,\min} = \min(\Delta E_{C,i}) \quad (1009)$$

This is incorporated into Sentaurus Device as a strain-dependent electron effective mass using:

$$m'_n = \gamma_n^{2/3} \cdot m_n \quad (1010)$$

so that:

$$\dot{N}_C = \frac{m'_n}{m_n}^{3/2} \cdot N_C \quad (1011)$$

For materials such as Ge and III–V, the stress-related change of the effective DOS should account for L- and  $\Gamma$ -electron valleys as well. The stress effect in L- and  $\Gamma$ -electron valleys is described by the linear deformation potential model (Equation 995) where the valley effective mass change is not accounted for. This simplifies the model where L- and  $\Gamma$ -valleys are added to  $\gamma_n$  similarly as suggested by Equation 1008:

$$\gamma_n = \frac{1}{N_C} \cdot \sum_{i=1}^3 N_{\Delta_2,i} \cdot e^{\frac{\Delta E_{C,\min} - \Delta E_{\Delta_2,i}}{kT_n}} + \sum_{i=1}^4 N_{L,i} \cdot e^{\frac{\Delta E_{C,\min} - \Delta E_{L,i}}{kT_n}} + N_{\Gamma} \cdot e^{\frac{\Delta E_{C,\min} - \Delta E_{\Gamma}}{kT_n}} \quad (1012)$$

where:

- $N_{\Delta_2,i}$  is the effective DOS of the  $\Delta_2$ -valley (corresponds to  $N_{C,i}$  in Equation 1006).
- $N_{L,i}$  is the effective DOS of each L-valley.
- $N_{\Gamma}$  is the effective DOS of each  $\Gamma$ -valley.
- The  $\Delta_2$ -valley, L-valley, and  $\Gamma$ -valley energy shifts  $\Delta E_{\Delta_2,i}$ ,  $\Delta E_{L,i}$ ,  $\Delta E_{\Gamma}$  are in reference to the conduction band edge.
- $\Delta E_{C,\min}$  is the minimum between all energy shifts  $\Delta E_{\Delta_2,i}$ ,  $\Delta E_{L,i}$ ,  $\Delta E_{\Gamma}$  (as in Equation 1009).
- $N_C$  in Equation 1012 is the relaxed effective DOS of the conduction band, which is computed with an account of all  $\Delta_2$ -, L-, and  $\Gamma$ -valleys.

## Chapter 31: Mechanical Stress

Deformation of Band Structure

### Strained Hole Effective Mass and DOS

To compute the hole effective DOS mass for arbitrary strain in silicon, the band structure provided by the six-band  $k \cdot p$  method is explicitly integrated assuming Boltzmann statistics [14][15]. In this approximation, the total hole effective DOS mass is given by:

$$m'_p(T) = [m_{cc,1}^{3/2} e^{-(E_3 - E_1)/(kT)} + m_{cc,2}^{3/2} e^{-(E_3 - E_2)/(kT)} + m_{cc,3}^{3/2}]^{2/3} \quad (1013)$$

where  $E_1$ ,  $E_2$ , and  $E_3$  are the ordered band edges for each of the three valence valleys, and  $m_{cc,1}$ ,  $m_{cc,2}$ , and  $m_{cc,3}$  are the carrier-concentration masses for each of the valleys given by:

$$m_{cc}^{3/2}(T) = \frac{2}{\sqrt{\pi}} (kT)^{-3/2} \int_0^{\infty} dE m_{DOS}^{3/2}(E) \sqrt{E} e^{-E/(kT)} \quad (1014)$$

The energy-dependent DOS mass of each valley,  $m_{DOS}$ , is given by:

$$m_{DOS}^{3/2}(E) = \frac{\sqrt{2}\pi^2\hbar^3}{\sqrt{E}} \frac{1}{(2\pi)^3} \int_0^{2\pi} d\phi \int_0^{\pi} d\theta \sin(\theta) \left[ \left| \frac{\partial k}{\partial E} \right| k^2 \right]_{\phi, \theta, E} \quad (1015)$$

The band structure-related integrand is computed from the inverse six-band  $k \cdot p$  method in polar  $k$ -space coordinates. The integrals in Equation 1014 and Equation 1015 are evaluated using optimized quadrature rules.

The six-band  $k \cdot p$  method is controlled by seven parameters:

- Three Luttinger–Kohn parameters ( $\gamma_1$ ,  $\gamma_2$ ,  $\gamma_3$ ) that determine the band dispersion.
- The spin-orbit split-off energy ( $\Delta_{so}$ ).
- Three deformation potentials (a, b, d) that determine the strain response.

The Luttinger–Kohn parameters and  $\Delta_{so}$  have been set to reproduce the DOS mass for relaxed silicon,  $m_p$ , as given by Equation 180 as a function of temperature. The default deformation potentials have been taken from the literature [16].

The strain-dependent valence-band effective DOS is then calculated from:

$$N'_V = \gamma_p \cdot N_V \quad (1016)$$

where  $\gamma_p = \begin{bmatrix} m_p^{3/2} & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}$ .

## Chapter 31: Mechanical Stress

Deformation of Band Structure

### Using Strained Effective Masses and DOS

The strain-dependent effective mass and DOS calculations can be selected by specifying `DOS(eMass)`, `DOS(hMass)`, or `DOS(eMass hMass)` as an argument to `Piezo(Model())` in the `Physics` section of the command file.

For example:

```
Physics {
    Piezo (
        Model (
            DOS (eMass hMass)
        )
    )
}
```

These models have been calibrated only for strained silicon, germanium, SiGe, and several III-V materials. Most of the model parameters are defined in the `LatticeParameters` section of the parameter file. Parameters affecting the `DOS(eMass)` model for  $\Delta_2$ -valleys include the deformation potentials of the k·p model for electron bands (`xis`, `dbs`, `xiu`, and `xid`) and the Sverdlov k·p parameter (`MKP`). Parameters affecting the `DOS(eMass)` model for L-valleys include the deformation potentials (`xiu_l` and `xid_l`), the relaxed energy difference between  $\Delta_2$ -valleys and L-valleys (`e_l`), and the effective masses (`me_10_l` and `me_10_t`) defined in the `StressMobility` section (the masses define the effective DOS of one L-valley  $N_{L,i}$  in [Equation 1012](#)). Parameters affecting the `DOS(eMass)` model for  $\Gamma$ -valleys include the deformation potential (`xid_gamma`), the relaxed energy difference between  $\Delta_2$ - and  $\Gamma$ -valleys (`e_gamma`), and the effective mass (`me0_gamma`) defined in the `StressMobility` section (the mass defines the effective DOS of one  $\Gamma$ -valley  $N_{\Gamma}$  in [Equation 1012](#)). Parameters affecting the `DOS(hMass)` model include the deformation potentials of the k·p model for hole bands (`adp`, `bdp`, `ddp`, `dso`) and the Luttinger parameters (`gamma_1`, `gamma_2`, `gamma_3`). These parameters can be specified in the `LatticeParameters` section of the parameter file and can be mole fraction dependent.

The `DOS(hMass)` model involves stress-dependent and lattice temperature-dependent numeric integrations that can be very CPU intensive. For simulations at a constant lattice temperature, you should only observe this CPU penalty once, usually at the beginning of the simulation. For nonisothermal thermal simulations, these integrations would usually have to be repeated for every lattice temperature change during the solution, resulting in a very prohibitive simulation.

As an alternative, Sentaurus Device provides an option for modeling the lattice temperature dependency of the strain-affected hole mass with analytic expressions that are fit to the full numeric integrations for each stress in the device. A CPU penalty will still be observed at the beginning of the simulation while fitting parameters are determined, but the remainder of the simulation should proceed as usual since the analytic expression evaluations are very fast.

## Chapter 31: Mechanical Stress

### Deformation of Band Structure

By default, the analytic lattice temperature fit is used with thermal simulations and numeric integration is used for isothermal simulations. These defaults can be overridden by using the `AnalyticLTFit` or `NumericalIntegration` options for `DOS(hMass)`:

```
DOS(eMass hMass(NumericalIntegration))
DOS(eMass hMass(AnalyticLTFit))
```

---

## Multivalley Band Structure

The multivalley model is an alternative that accounts for the carrier population in various valleys presented in the semiconductor band structure where the stress effect is accounted for in each valley separately. Therefore, this model does not require to have effective corrections applied to the conduction and valence band edge-energy and DOS. Based on such detailed consideration of the carrier repopulation between valleys, this model gives a possibility to account for the stress effect in both the carrier density and mobility consistently.

For a general description of the multivalley model and its implementation, see [Multivalley Band Structure on page 327](#). The multivalley model accounts for the stress effect in the band structure by stress-induced change of the energy and effective masses in each valley.

The stress-induced change in the valley energy is described in [Deformation of Band Structure on page 940](#). It is accounted for in both  $k \cdot p$  bands [12][13] and arbitrary valleys defined in the parameter file (see [Using Multivalley Band Structure on page 331](#)).

The stress-induced change in the valley effective mass is accounted for in the two-band  $k \cdot p$  model [13] for electrons and the  $6 \times 6 k \cdot p$  model [12] for holes by a computation of the effective DOS factors  $g_{n,i}$ ,  $g_{p,i}$  in [Equation 195](#) and [Equation 196](#). The stress-induced change of the effective DOS masses for each valley is described in [Strained Effective Masses and Density-of-States on page 945](#).

Referring to [Equation 1006](#) and [Equation 1013](#), and using their variables, the effective DOS factors of each  $\Delta_2$  valley can be expressed as follows if only  $k \cdot p$  bands are defined in the multivalley model:

$$g_{n,i} = \frac{1}{3} \sqrt{\frac{m_{t1,i}}{m_t} \frac{m_{t2,i}}{m_t} \frac{m_{t3,i}}{m_t}} \quad \quad g_{p,i} = \left[ \frac{m_{cc,i}}{m_p} \right]^{3/2} \quad (1017)$$

For a general case, where both  $k \cdot p$  bands and arbitrary valleys are defined, the effective DOS factors are computed similarly to [Equation 208 on page 335](#).

In most III–V materials, electron transport in the  $\Gamma$ -valley must be accounted for properly. Usually, this valley is strongly nonparabolic and its properties are affected by the stress. According to the  $k \cdot p$  perturbation theory [17], the electron  $\Gamma$ -valley mass  $m_\Gamma$  can be represented as  $m_0/m_\Gamma - 1 = (P^2/3)(2/E_\Gamma + 1/(E_\Gamma + \Delta_0))$  where:

- $E_\Gamma$  is the  $\Gamma$ -valley energy in reference to the valence band energy.
- $m_0$  is the free electron mass.

## Chapter 31: Mechanical Stress

### Deformation of Band Structure

- $\Delta_0$  is the valence band spin-orbit splitting energy.
- $P^2$  is the squared conduction-to-valence momentum matrix element.

An assumption that  $P^2$  does not depend on the stress gives the following expression for the stress-dependent  $\Gamma$ -valley mass  $m'_\Gamma$ :

$$R_m = \frac{\frac{2}{E_\Gamma} + \frac{1}{E_\Gamma + \Delta_0}}{\frac{2}{E_\Gamma} + \frac{1}{E_\Gamma + \Delta_0}} - 1 \quad A_m + 1 \quad (1018)$$

$$\frac{m'_\Gamma}{m_0} = \frac{R_m}{\frac{m_0}{m_\Gamma} + R_m - 1}$$

where  $E'_\Gamma = E_\Gamma + \Delta E_\Gamma$ ,  $\Delta E_\Gamma$  is the stress-related energy shift of the  $\Gamma$ -valley,  $A_m$  is a fitting parameter where, if  $A_m = 0$ , there is no stress effect in the  $\Gamma$ -valley mass.

Similarly, based on [18], the stress-dependent band nonparabolicity  $\alpha'_\Gamma$  of the  $\Gamma$ -valley could be expressed as follows:

$$R_\alpha = \frac{\frac{1}{E_\Gamma} - \frac{m'_\Gamma}{m_0}^2}{\frac{1}{E_\Gamma} - \frac{m'_\Gamma}{m_0}} - 1 \quad A_\alpha + 1 \quad (1019)$$

$$\alpha'_\Gamma = \alpha_\Gamma R_\alpha$$

where  $\alpha_\Gamma$  is the relaxed band nonparabolicity of the  $\Gamma$ -valley,  $A_\alpha$  is a fitting parameter where, if  $A_\alpha = 0$ , there is no stress effect in the  $\Gamma$ -valley nonparabolicity.

## Using Multivalley Band Structure

The multivalley model is activated with the keyword `Multivalley` in the `Physics` section. If the model must be activated only for electrons or holes, the keywords `eMultivalley` and `hMultivalley` can be used. All options of the multivalley model can be used in stress simulations (see [Using Multivalley Band Structure on page 331](#)).

### Note:

Although the multivalley model works together with any `DeformationPotential` model (it recomputes all valley energy shifts with reference to the band edges defined by the deformation potential model), Sentaurus Device stops with an error message if the `DOS` statement is used (see [Strained Effective Masses and Density-of-States on page 945](#)).

For the carrier density computation, all parameters of arbitrary valleys can be changed in the `Multivalley` section of the parameter file (see [Using Multivalley Band Structure on](#)

## Chapter 31: Mechanical Stress

### Mobility Modeling

page 331). However, for  $k \cdot p$  bands, you should use the `LatticeParameters` section of the parameter file (see [Using Deformation Potential Model on page 943](#)).

In III-V materials, the band nonparabolicity plays an important role in the carrier density and mobility. To account for it in the multivalley model, use `MultiValley(Nonparabolicity)` in the `Physics` section. To activate the stress-dependent  $\Gamma$ -valley mass and the nonparabolicity models, as in [Equation 1018](#) and [Equation 1019](#), the keywords `m0` and `alpha0` must be used instead of `m` and `alpha` in the valley specification (see [Using Multivalley Band Structure on page 331](#)). Such an option is possible only for valleys with isotropic mass. For example, for InAs, it could be as follows:

```
eValley"Gamma" (m0=0.0244 energy=0.0 alpha0=1.39 degeneracy=1 xid=-10.2)
```

The fitting parameters  $A_m$  and  $A_\alpha$  from [Equation 1018](#) and [Equation 1019](#) can be specified globally for all valleys as follows in the `Multivalley` section of the parameter file:

```
Multivalley{  
    BandgapMassFactor = Am  
    BandgapAlphaFactor = Aα  
}
```

The default value of both the fitting parameters  $A_m$  and  $A_\alpha$  is equal to 1.

The multivalley band structure can be used to compute the stress-induced change in the carrier mobility with the advanced mobility models, which account for the carrier interface quantization in each valley (see [Multivalley Electron Mobility Model on page 953](#) and [Multivalley Hole Mobility Model on page 963](#)). For these models, additional band structure-related parameters can be changed in the `StressMobility` section of the parameter file. To have better flexibility and to use other than the multivalley MLDA quantization model in the carrier density (see [Chapter 14 on page 348](#)), there is an option to exclude the multivalley model from the carrier density computation, but still have it in the stress-induced mobility models. For that, use `Multivalley(-Density)`.

---

## Mobility Modeling

The presence of mechanical stress in device structures results in anisotropic carrier mobility that must be described by a mobility tensor. The electron and hole current densities under such conditions are given by:

$$\hat{\vec{J}}_\alpha = \frac{\bar{\mu}_\alpha}{\bar{\mu}_{\alpha\bar{0}}} \vec{J}_{\alpha\bar{0}}, \quad \alpha = n, p \quad (1020)$$

where:

- $\bar{\mu}_\alpha$  is the stress-dependent mobility tensor.

## Chapter 31: Mechanical Stress

### Mobility Modeling

- $\mu_{\alpha 0}$  denotes the isotropic mobility without stress (the reference mobility in the specific transport direction).
- $\vec{J}_{\alpha 0}$  is the carrier current density without stress.

#### Note:

[Equation 1020](#) is a general expression that is used to correct the carrier current density  $\vec{J}_{\alpha 0}$ . The mobility model corrections  $\mu_{\alpha}/\mu_{\alpha 0}$  described in this section account for detailed semiconductor band structure, interface and channel orientations effects, interface and geometric quantizations, and so on. Therefore, these models are not limited to only stress problems and can be used in other applications where such effects are important. As a result, not related to stress problems, if the geometric quantization model is used (with the `ThinLayer` keyword; see [Nonparabolic Bands and Geometric Quantization on page 372](#)), the relative isotropic mobility  $\mu_{\alpha 0}$  is computed without the geometric quantization effect. This allows you to account for a dependency of the carrier current density  $\vec{J}_{\alpha}$  on the layer thickness due to the geometric quantization effect.

The following sections describe options available in Sentaurus Device for including the effects of stress, interface or channel orientation, and geometric confinement in thin layers on the carrier mobilities  $\mu_{\alpha}$  and  $\mu_{\alpha 0}$ .

#### Note:

For a general case that does not require you to consider any specific transport direction in the reference mobility, you can use a full tensor reference mobility  $\underline{\mu}_{\alpha 0}$ . In this case, [Equation 1020](#) must be rewritten as follows  
$$\vec{J}_{\alpha} = \mu_{\alpha} \times (\underline{\mu}_{\alpha 0})^{-1} \vec{J}_{\alpha 0}$$
. Such an option can be time consuming, specially for holes.

---

## Multivalley Electron Mobility Model

To calculate stress-induced electron mobility, Sentaurus Device considers several approaches [\[6\]](#)[\[13\]](#)[\[20\]](#)[\[21\]](#) for the mobility approximation, and it computes the mobility ratio ( $3 \times 3$  tensor), which corrects the relaxed current density in [Equation 1020](#). The simplest approach [\[6\]](#) focuses on the modeling of the mobility changes due to the carrier redistribution between bands in silicon. As a known example, the electron mobility is enhanced in a strained-silicon layer grown on top of a thick, relaxed SiGe layer. Due to the lattice mismatch (which can be controlled by the Ge mole fraction), the thin silicon layer appears to be ‘stretched’ (under biaxial tension).

The origin of the electron mobility enhancement can be explained [\[6\]](#) by considering the six-fold degeneracy in the conduction band. The biaxial tensile strain lowers two perpendicular valleys ( $\Delta_2$ ) with respect to the four-fold in-plane valleys ( $\Delta_4$ ). Therefore, electrons are redistributed between valleys and  $\Delta_2$  is occupied more heavily. It is known

## Chapter 31: Mechanical Stress

### Mobility Modeling

that the perpendicular effective mass is much lower than the longitudinal one. Therefore, this carrier redistribution and reduced intervalley scattering enhance the electron mobility.

The model consistently accounts for a change of band energy as described in [Deformation of Band Structure on page 940](#), that is, a modification of the deformation potentials in [Equation 954](#) will affect the strain-silicon mobility model. In the crystal coordinate system, the model gives only the diagonal elements of the electron mobility matrix.

Based on the model [6] that accounts only for carrier occupation, the following expressions have been suggested for the electron mobility:

$$\mu_{n,ii} = \mu_{n0} \left[ 1 + \frac{1 - m_{n1}/m_{nt}}{1 + 2(m_{n1}/m_{nt})} \frac{\frac{F_n - E_C - \Delta E_{C,i}}{kT}}{\frac{F_n - E_C - \Delta E_C}{kT}} - 1 \right] \quad (1021)$$

where:

- $\mu_{n0}$  is electron mobility without the strain.
- $m_{n1}$  and  $m_{nt}$  are the electron longitudinal and transverse masses in the subvalley, respectively.
- $\Delta E_{C,i}$  and  $\Delta E_C$  are computed by [Equation 954](#) and [Equation 997](#) or [Equation 998](#), respectively.
- The index  $i$  corresponds to a direction (for example,  $\mu_{n,11}$  is the electron mobility in the direction of the x-axis of the crystal system and, therefore,  $\Delta E_{C,1}$  should correspond to the two-fold subvalley along the x-axis).
- $F_n$  is quasi-Fermi level of electrons.

#### Note:

For the carrier quasi-Fermi levels, as for [Equation 1021](#), there are two options to be computed: (a) assuming the charge neutrality between carrier and doping, and it gives only the doping dependence of the model and (b) using the local carrier concentration (see [Using Multivalley Electron Mobility Model on page 960](#)).

Derivation of [Equation 1021](#) is based on consideration of the change in the stress-induced band energy in bulk silicon. However, in MOSFETs, there is an additional influence of a quantization that appears in the carrier channel at the silicon–oxide interface. For example, there is a reduction of the stress effect with applied gate voltage. Partially, this is due to the Fermi-level dependency on the carrier concentration, as well as to the quantization in the channel gives a different carrier redistribution between electron bands (see [Inversion Layer on page 958](#)).

## Intervalley Scattering

Another effect of the stress-induced mobility change is intervalley scattering, which is considered in [20] for the bulk case. According to that model, the total relaxation time in valley  $i$  is expressed as follows:

$$\frac{1}{\tau_i} = \frac{1}{\tau^g} + \frac{1}{\tau_i^f} + \frac{1}{\tau^I} \quad (1022)$$

where:

- $\tau^g$  denotes the momentum relaxation time due to acoustic intravalley scattering and intervalley scattering between equivalent valleys (g-type scattering).
- $\tau^I$  is the impurity scattering relaxation time.
- $\tau_i^f$  is the relaxation time for intervalley scattering between nonequivalent valleys (f-type scattering).

The intervalley scattering rate for electrons to scatter from initial valley  $i$  to final valley  $j$  can be defined as [20]:

$$S(\epsilon, \Delta_{ij}) = C \left[ (\epsilon - \Delta_{ij}^{\text{emi}})^{1/2} + \exp \left( \frac{\hbar \omega_{\text{opt}}}{kT} \right) (\epsilon - \Delta_{ij}^{\text{abs}})^{1/2} \right] \quad (1023)$$

$$\Delta_{ij}^{\text{emi}} = \Delta E_{C,j} - \Delta E_{C,i} - \hbar \omega_{\text{opt}}$$

$$\Delta_{ij}^{\text{abs}} = \Delta E_{C,j} - \Delta E_{C,i} + \hbar \omega_{\text{opt}}$$

where  $\hbar \omega_{\text{opt}}$  is the phonon energy and  $C$  is a constant. The relaxation time for this scattering event can be defined as follows:

$$\frac{1}{\tau_f(i \rightarrow j)} = \sum_0^\infty S(\epsilon, \Delta_{ij}) f(\epsilon, F_n) d\epsilon \quad (1024)$$

where  $f(\epsilon, F_n)$  is the electron distribution function.

Using the Fermi–Dirac distribution function and considering that electrons from the valley  $i$  can be scattered into two valleys  $j$  and  $l$  ( $1/\tau_i^f = 1/\tau^f(i \rightarrow j) + 1/\tau^f(i \rightarrow l)$ ), a ratio between unstrained and strained relaxation times for this intervalley scattering in valley  $i$  can be written as:

$$h_i = \frac{\tau_i^{f0}}{\tau_i^f} = \frac{F_{1/2} \frac{\eta - \Delta_{ij}^{\text{emi}}}{kT} + F_{1/2} \frac{\eta - \Delta_{il}^{\text{emi}}}{kT} + \exp \left( \frac{\hbar \omega_{\text{opt}}}{kT} \right) \left[ F_{1/2} \frac{\eta - \Delta_{ij}^{\text{abs}}}{kT} + F_{1/2} \frac{\eta - \Delta_{il}^{\text{abs}}}{kT} \right]}{2 \left[ F_{1/2} \frac{\eta + \hbar \omega_{\text{opt}}}{kT} + \exp \left( \frac{\hbar \omega_{\text{opt}}}{kT} \right) F_{1/2} \frac{\eta - \hbar \omega_{\text{opt}}}{kT} \right]} \quad (1025)$$

where  $\eta = F_n - E_C$ .

## Chapter 31: Mechanical Stress

### Mobility Modeling

#### Note:

The authors in [20] used the Boltzmann distribution function to express  $h_i$ . As a result, Eq. 20 of [20] does not have any carrier concentration (or doping) dependence, but [Equation 1025](#) does have it through the Fermi level  $F_n$ .

Considering undoped/doped and unstrained/strained cases, the paper [20] derives an expression for total mobility change in valley  $i$ , which is based on a doping-dependent mobility model. With simplified doping dependence, a ratio between strained and unstrained total relaxation times for the valley  $i$  can be expressed as follows:

$$\frac{\tau_i}{\tau_0} = \frac{1}{1 + (h_i - 1) \frac{1 - \beta^{-1}}{1 + \frac{N_{\text{tot}}}{N_{\text{ref}}} \alpha}} \quad (1026)$$

where  $N_{\text{tot}}$  is the sum of donor and acceptor impurities, and  $\beta = 1 + \tau^g / \tau^f$  is a fitting parameter [20] as both others  $N_{\text{ref}}$  and  $\alpha$ .

The final modification of the mobility along valley  $i$  (in [Equation 1021](#)), which includes the stress-induced carrier redistribution and change in the intervalley scattering, is:

$$\mu_{n, ii} = \frac{3\mu_{n0}}{1 + 2(m_{n1}/m_{n0})} \cdot \frac{\frac{\tau_i}{\tau_0} F_{1/2} \frac{n - \Delta E_{C,i}}{kT} + \frac{\tau_i m_{n1}}{\tau_0 m_{n0}} F_{1/2} \frac{n - \Delta E_{C,i}}{kT} + \frac{\tau_i m_{n1}}{\tau_0 m_{n0}} F_{1/2} \frac{n - \Delta E_{C,f}}{kT}}{F_{1/2} \frac{n - \Delta E_{C,i}}{kT} + F_{1/2} \frac{n - \Delta E_{C,f}}{kT} + F_{1/2} \frac{n - \Delta E_{C,f}}{kT}} \quad (1027)$$

#### Note:

The model ([Equation 1025](#) and [Equation 1026](#)) was developed originally for the bulk case. However, in MOSFET channels, to obtain an agreement with experimental data, the model and parameter  $\beta$ , responsible for the unstressed ratio between g-type and f-type scatterings, must be modified (see [Inversion Layer on page 958](#)).

## Effective Mass

It is known that the effective mass of electrons does not change significantly if the stress is applied along the crystal axis, and it allows you to write the stress-induced mobility change in the form of [Equation 1021](#) and [Equation 1027](#). However, an analysis based on the empirical nonlocal pseudopotential theory [21] shows that, for the stresses applied along  $\langle 110 \rangle$ , the effective mass changes strongly and it affects the electron mobility. The literature [21] suggests a simple empirical polynomial approximation for the dependency of the effective mass on stress applied along  $\langle 110 \rangle$ . Later, the two-band k·p theory [13] was developed for electrons with the main analytic results for the effective masses described in [Strained Electron Effective Mass and DOS on page 945](#).

## Chapter 31: Mechanical Stress

### Mobility Modeling

To formulate the stress-induced change of the mobility generally (accounting for arbitrary channel and interface orientations), the inverse conductivity mass tensor for the valley  $i$  is represented in the following very general form:

$$(\bar{m}_{\text{cond}, i})^{-1} = \frac{1}{\hbar^2} \frac{\partial^2 \vec{\varepsilon}^i(\vec{k})}{\partial k_j \partial k_l} \quad (1028)$$

where:

- $\vec{\varepsilon}^i(\vec{k})$  is the energy dispersion of valley  $i$  defined by the two-band k·p theory.
- $k_j, k_l$  are the wavevectors.
- The inverse mass tensor  $(\bar{m}_{\text{cond}, i})^{-1}$  is computed in the valley energy minima, which is a good approximation for electron conductivity masses in silicon.

Near an interface, you can assume zero current perpendicular to the interface. This assumption and a consideration of the carrier quantization near the interface [22] suggest that the  $3 \times 3$  inverse conductivity mass tensor of Equation 1028 should be recomputed into a  $2 \times 2$  in-plane tensor. For the inverse conductivity mass tensor in the interface coordinate system (with components  $w_{ij}$ ) where the perpendicular axis has an index 3, the following recomputation into the  $2 \times 2$  in-plane symmetric tensor should be performed:

$$\begin{aligned} w'_{11} &= w_{11} - \frac{w_{13}^2}{w_{33}} \\ w'_{22} &= w_{22} - \frac{w_{23}^2}{w_{33}} \\ w'_{12} &= w_{12} - \frac{w_{13} w_{23}}{w_{33}} \end{aligned} \quad (1029)$$

Setting  $w'_{13} = w'_{23} = 0$  and  $w'_{33} = w_{33}$ , this tensor is rotated back to the crystal coordinate system and is used as  $(m_{\text{cond}, i})^{-1}$ . Such a recomputation has an effect for cases where the valley ellipsoid in k-space is not aligned to the interface.

Another optional mass transformation can be performed for 1D carrier transport that appears, for example, in nanowires. The 1D mass transformation is performed in the vicinity of the interface and with a specified direction of the 1D carrier transport. This direction (the user-defined vector in the simulation coordinate system) is projected on to the interface plane. The projected in-plane vector and the normal vector to the interface define a coordinate system where the tensor from Equation 1029 (with 2D transport mass transformation) is rotated.

## Chapter 31: Mechanical Stress

### Mobility Modeling

Assuming that the index 1 of the rotated tensor is along the 1D transport direction, the corresponding inverse mass tensor component can be written as:

$$w''_{11} = w'_{11} - \frac{(w'_{12})^2}{w'_{22}} \quad (1030)$$

Similar to [Equation 1029](#), when setting  $w''_{12} = w''_{13} = w''_{23} = 0$  and  $w''_{22} = w'_{22}$ ,  $w''_{33} = w'_{33}$ , this diagonal tensor is rotated back to the crystal coordinate system and is used as  $(\bar{m}_{\text{cond}, i})^{-1}$ .

To account for such stress-induced mass change, [Equation 1021](#) and [Equation 1027](#) are generalized as stated in the next section, [Inversion Layer](#).

## Inversion Layer

Generally, using only the valley data, the total mobility tensor could be written in the following general form, which generalizes [Equation 1021](#), and [Equation 1027](#):

$$\bar{\mu}_n = q\tau_0 \sum_i \frac{n_i \tau_i}{n \tau_0} (\bar{m}_{\text{cond}, i})^{-1} \quad (1031)$$

where:

- $n_i/n$  is a local valley occupation.
- $\tau_i/\tau_0$  is the ratio of stressed and unstressed relaxation times as it is in [Equation 1026](#).
- $(\bar{m}_{\text{cond}, i})^{-1}$  is the inverse conductivity mass tensor [Equation 1028](#).

To use [Equation 1031](#) in Sentaurus Device stress modeling, the mobility tensor  $\bar{\mu}_n$  should be divided by unstressed mobility  $\mu_{n0}$  because the stress effect is accounted for as a multiplication factor to TCAD mobility (for example, the Lombardi model) used in the device simulation. Therefore, the unstressed mobility  $\mu_{n0}$  also is computed by [Equation 1031](#), but with zero stress.

In the multivalley representation, [Equation 1031](#) is general enough to describe the mobility for both bulk and MOSFET inversion layer cases, but a difference between these cases is in the  $n_i/n$  and  $\tau_i/\tau_0$  terms. For inversion layers, the quantization effect must be accounted for in each valley separately.

At this point, only the multivalley MLDA model gives such a possibility (see [Modified Local-Density Approximation Model on page 370](#)). Therefore, for the inversion layer mobility, the local valley occupation is computed by the multivalley MLDA model where the quantization mass  $m_q$  in the perpendicular direction to the interface is computed using stressed k·p bands (by a rotation of the inverse mass tensor  $(\bar{m}_{\text{cond}, i})^{-1}$  to an interface coordinate system).

## Chapter 31: Mechanical Stress

### Mobility Modeling

#### Note:

The multivalley MLDA model accounts for arbitrary interface orientation automatically (see [Interface Orientation and Stress Dependencies on page 370](#)), which complements the general inverse mass tensor  $(\bar{m}_{\text{cond}, i})^{-1}$  in [Equation 1028](#), and it gives you an automatic option that accounts for both channel and interface orientations simultaneously.

For the inversion layer mobility model, there are some changes to the scattering ratio  $\tau_i/\tau_0$  in [Equation 1025](#). First, the DOS used in [Equation 1023](#) is a parabolic bulk DOS, which makes such a scattering model mostly applicable to the bulk case. The MLDA quantization model ([Equation 240](#)) gives an MLDA DOS  $D'(\varepsilon, z)$  that becomes the bulk DOS at a distance from the interface where the quantum effect is small. Second, [Equation 1023](#) accounts only for the intervalley optical phonon scattering, but [Equation 1032](#) includes both intervalley and intravalley scattering with acoustic and optical phonons.

In addition, this equation takes into account an effect of the interface quantization in the scattering by the following use of both MLDA and bulk DOS:

$$\begin{aligned} \frac{j}{\tau_{\text{ac}}^i(\varepsilon, z)} &= \frac{2\pi k T}{\hbar \rho} \frac{D_{\text{ac}}^2}{c_1} \sum_j \sum_k D_{\text{bulk}}^j(\varepsilon) D_{\text{bulk}}^k(\varepsilon - \Delta\varepsilon_0^{k,i}) \\ \frac{j}{\tau_{\text{ope}}^i(\varepsilon, z)} &= \frac{\pi \hbar D_{\text{op}}^2}{\rho \hbar \omega_{\text{opt}}} (N_{\text{op}} + 1) \sum_j \sum_k D_{\text{bulk}}^j(\varepsilon) D_{\text{bulk}}^k(\varepsilon - \hbar \omega_{\text{opt}} - \Delta\varepsilon_0^{k,i}) \\ \frac{j}{\tau_{\text{opa}}^i(\varepsilon, z)} &= \frac{\pi \hbar D_{\text{op}}^2}{\rho \hbar \omega_{\text{opt}}} N_{\text{op}} \sum_j \sum_k D_{\text{bulk}}^j(\varepsilon) D_{\text{bulk}}^k(\varepsilon + \hbar \omega_{\text{opt}} - \Delta\varepsilon_0^{k,i}) \\ S^i(\varepsilon, z) &= \frac{1}{\tau_{\text{ac}}^i(\varepsilon, z)} + \frac{1}{\tau_{\text{ope}}^i(\varepsilon, z)} + \frac{1}{\tau_{\text{opa}}^i(\varepsilon, z)} \end{aligned} \quad (1032)$$

where:

- $D_{\text{bulk}}^i(\varepsilon) = \frac{2}{(2\pi)^3} \circ \frac{dS}{\left| \nabla_k \varepsilon^i(\vec{k}) \right|}$  is the electron bulk DOS of the valley  $i$ .
- $\Delta\varepsilon_0^{k,i} = \Delta E_{C,k} - \Delta E_{C,i}$  is a valley minimum energy difference between valleys  $i$  and  $k$ .
- $\frac{D_{\text{ac}}}{c_1}$  is the ratio of the acoustic-phonon deformation potential and the sound velocity.
- $\rho$  is the mass density.
- $\hbar \omega_{\text{opt}}$  is the optical-phonon energy.

## Chapter 31: Mechanical Stress

### Mobility Modeling

- $D_{\text{op}}$  is the optical-phonon deformation potential.
- $N_{\text{op}} = [\exp(\hbar\omega_{\text{opt}}/kT) - 1]^{-1}$  is the phonon number.

#### Note:

For the bulk case,  $D^j(\epsilon, z) = D_{\text{bulk}}^j(\epsilon)$  and, therefore, [Equation 1032](#) can be simplified to regular bulk scattering rate equations. [Equation 1032](#) accounts for both interband and intraband phonon scattering with the same deformation potentials, but there is an option to control the interband scattering factor.

Using [Equation 1032](#) will activate a numeric integration in [Equation 1024](#) with Gauss–Laguerre quadratures as described in [Nonparabolic Band Structure on page 328](#). Another change in the scattering ratio  $\tau_i/\tau_0$  is based on comparisons to various stress- and orientation-dependent experimental data. It gives the new user-defined parameter  $\beta$  for the inversion layer (see [Using Multivalley Electron Mobility Model on page 960](#)).

For the bulk case, the unstressed mobility  $\mu_{n0}$  computed by [Equation 1031](#) is always isotropic. However, for the MOSFET on a (110) silicon substrate, it is not. In this case, [Equation 1031](#) without stress gives anisotropic unstressed electron mobility, and a ratio between the mobilities of the <100> and <110> channel orientations is equal to approximately 1.1. This ratio is in reasonable agreement to experimentally observed data where the ratio changes from 1.2 to 1.1 for a high-gate electric field.

#### Note:

By default, the unstressed mobility  $\mu_{n0}$  (in [Equation 1020](#)) is always computed for a <110> channel orientation and for a substrate orientation that corresponds to the auto-orientation Lombardi option (see [Auto-Orientation for the Lombardi Model on page 407](#)). Such a default requires you to have calibrated the Lombardi model parameters for the <110> channel (critical for MOSFETs on (110) substrate where the unstressed mobility is anisotropic).

## Using Multivalley Electron Mobility Model

The stress-induced mobility model can be activated regionwise or materialwise. To activate the inversion layer model (see [Inversion Layer on page 958](#)), the `eMultivalley(MLDA)` statement must be specified in the `Physics` section. In this case, all valleys defined by the multivalley model (for example, with `eMultivalley(kpDOS parfile)`; see [Multivalley Band Structure on page 327](#)) will be used in the model. However, it is calibrated for silicon, germanium, and SiGe band structures (with  $\Delta_2$ - and L-valleys), and for InGaAs materials where the  $\Gamma$ -valley carrier transport is mostly important.

The keyword `MLDA` activates the multivalley MLDA quantization model (see [Modified Local-Density Approximation Model on page 370](#)), which is applied to both the density and the inversion layer stress-induced mobility models. To use another quantization model in the density computation, specify the `-Density` option in the `eMultivalley` statement (see [Notes on the Use of the MLDA Model on page 377](#)).

## Chapter 31: Mechanical Stress

### Mobility Modeling

To ensure that electron stress-related models are consistent and physically correct, the Physics section should contain the following:

```
Physics {
    eMultiValley(MLDA)
    Piezo(Model(
        Mobility(eSubband(Fermi EffectiveMass Scattering(MLDA)))
    )))
}
```

#### Note:

If the model is used without options, as eSubband, then the recommended options are activated, and this is equivalent to eSubband(Doping EffectiveMass Scattering(MLDA)).

The keyword Fermi defines that the Fermi–Dirac distribution function is used in the mobility models [Equation 1021](#), [Equation 1027](#), and [Equation 1031](#) with the carrier self-consistent quasi-Fermi energy. To activate only doping-dependent quasi-Fermi energy, use eSubband(Doping). If neither Fermi nor Doping is used, the Boltzmann distribution function is used, which illuminates the quasi-Fermi energy dependency.

To activate the model for a stress-induced change of the electron effective mass (see [Effective Mass on page 956](#), [Equation 1028](#)), use the keywords Mobility(eSubband(EffectiveMass)). If this keyword is not present, the unit diagonal (and stress-independent) tensor is used as  $(m_{\text{cond}, i})$  in [Equation 1031](#).

Using EffectiveMass(-Transport) excludes the 2D inverse conductivity mass tensor recomputation ([Equation 1029](#)). With the option EffectiveMass(Transport<vector>), you can activate the 1D transport mass recomputation ([Equation 1030](#)) where <vector> represents the direction of 1D carrier transport in the simulation coordinate system.

The unstressed longitudinal and perpendicular effective masses  $m_l$  and  $m_t$  for this model (to be used in two-band k·p theory; see [Strained Electron Effective Mass and DOS on page 945](#)) are defined as follows:

```
StressMobility {
    me_l0 = 0.914      # [1]
    me_t0 = 0.196      # [1]
}
```

Usually for III–V materials, the two-band k·p model for  $\Delta_2$ -valleys should not be used, and all valleys must be defined in the Multivalley section of the parameter file with all needed masses, energy shifts, deformation potentials, and so on. For some III–V materials such as InGaAs, GaAs, and InAs, all valleys of the semiconductor band structure are defined by the default in the parameter file. For such materials (where the electron  $\Gamma$ -valley is the lowest), the band nonparabolicity, the stress-induced  $\Gamma$ -valley mass change, and the geometric quantization effects in thin layers are important.

## Chapter 31: Mechanical Stress

### Mobility Modeling

To activate the related models, the following statement must be used:

```
eMultiValley(MLDA Nonparabolicity ThinLayer)
```

See [Nonparabolic Bands and Geometric Quantization on page 372](#) and [Multivalley Band Structure on page 950](#).

The statement `Scattering(MLDA)` means that the scattering rate is computed with [Equation 1032](#). If the keyword `Scattering` is not present, then  $\tau_i/\tau_0$  is unit and stress independent in [Equation 1027](#) and [Equation 1031](#). The keyword `MLDA` in the `Scattering` statement defines that the MLDA DOS is used in the scattering rate as this is in [Equation 1032](#). However, using only the keyword `Scattering` (without `MLDA`) activates the simplified bulk scattering rate of [Equation 1023](#).

For III–V materials, the statement `Scattering(MLDA2)` might be used, which replaces the bulk DOS  $D_{\text{bulk}}^j(\varepsilon)$  by the MLDA DOS  $D^j(\varepsilon, z)$  in [Equation 1032](#).

The parameters of the intervalley scattering model can be specified in the same `StressMobility` section of the parameter file, as this is below for silicon:

```
StressMobility {  
    Ephonon = 0.06          # [eV]  
    Dop = 1.25e9            # [eV/cm]  
    Dac_c1 = 1.027e-5       # [eVs/cm]  
    beta = 1.22             # [1]  
    beta_mlda = (1.5,1.5,1.5) # [1]  
    Nref = 3e19              # [cm^-3]  
    alpha = 0.65             # [1]  
}
```

where `Ephonon` is  $\hbar\omega_{\text{opt}}$  in [Equation 1023](#) and [Equation 1032](#), `Dop` is  $D_{\text{op}}$ , and `Dac_c1` is  $D_{\text{ac}}/c_1$ , in [Equation 1032](#). The parameters `Nref` and `alpha` correspond to the impurity scattering parameters  $N_{\text{ref}}$ ,  $\alpha$  in [Equation 1026](#). The fitting parameter `beta` in [Equation 1026](#) is  $\beta$  for the bulk case, and `beta_mlda` defines  $\beta$  for the inversion layer case for three interface orientations (100), (110), and (111). This gives additional calibration flexibility to users. Although, as described for [Equation 1026](#), the parameter  $\beta$  was initially introduced specifically for silicon, but generally, it is an estimation of the ratio of the total scattering rate to the scattering rate of scattering events that do not depend on the stress and layer thickness in the device, with no stress and no geometric confinement.

#### Note:

The inversion layer mobility model ( $\bar{\mu}_n$  in [Equation 1020](#)) is designed to work with arbitrary channel and interface orientations. This orientation is simply defined by a specification of only the `x` and `y` simulation coordinate axes in reference to the crystal coordinate system in the `LatticeParameters` section (see [Using Deformation Potential Model on page 943](#)).

The reference isotropic mobility ( $\mu_{n0}$  in [Equation 1020](#)), by default, is computed for the `<110>` channel direction. Such a default requires you to have calibrated the Lombardi model parameters for the `<110>` channel (critical for (110) interface

## Chapter 31: Mechanical Stress

### Mobility Modeling

orientation where the unstressed mobility is anisotropic). However, if you want to use <100>/(110) Lombardi model parameters, the keyword `-RelChDir110` must be used inside the `eSubband` statement. For a general case of the tensor reference mobility ( $\bar{\mu}_{n0}$ ), the keyword `-AutoOrientation` must be used instead.

All parameters of the `StressMobility` model can be mole fraction dependent. You can check this in the SiGe material parameter file.

---

## Multivalley Hole Mobility Model

Similar to the electron stress-induced mobility model (Equation 1031), the total hole mobility tensor can be written in the following general form, which can be applied to both bulk and inversion layer cases [23]:

$$\bar{\mu}_p = q\tau_0 \frac{p_i \tau}{p \tau_0} (\bar{m}_{\text{cond}, i})^{-1} \quad (1033)$$

where:

- $p_i/p$  is a local hole-band occupation.
- $\tau/\tau_0$  is a ratio of the stressed and unstressed valence-band relaxation times.
- $(\bar{m}_{\text{cond}, i})^{-1}$  is the inverse conductivity hole mass tensor.

The model accounts for the six-band k·p hole band structure [14] in all of the above three terms of Equation 1033. In the case of the inversion layer, the model computes the six-band k·p MLDA DOS of each band as described in [Modified Local-Density Approximation Model on page 370](#) and, correspondingly, it affects all terms of Equation 1033. Finally, the model computes the mobility ratio (3 × 3 tensor), which corrects the relaxed current density in [Equation 1020](#).

## Effective Mass

The inverse mass tensor  $(\bar{m}_{\text{cond}, i})^{-1}$  of the band  $i$  in Equation 1033 is based on an averaging of the reciprocal mass tensor in  $\mathbf{k}$ -space and energy space. If the reciprocal mass tensor at any  $\mathbf{k}$ -vector is expressed as follows:

$$w_{jl}^i(\vec{k}) = \frac{1}{\hbar^2} \frac{\partial^2 \epsilon_i(\vec{k})}{\partial k_j \partial k_l} \quad (1034)$$

## Chapter 31: Mechanical Stress

### Mobility Modeling

then the averaging in  $\mathbf{k}$ -space is performed in accordance to the DOS computation ([Equation 243](#)), which for the inversion layer can be formulated as:

$$w_{jl}^i(\epsilon, z) = \frac{\frac{2}{(2\pi)^3} \circ \frac{w_{jl}^i(\vec{k})}{\epsilon^i(\vec{k}) = \epsilon} \left[ 1 - \exp^{-i2z\gamma_{kp}} \frac{k_j \frac{w_{j3}(\vec{k})}{w_{33}(\vec{k})}}{D^i(\epsilon, z)} \right] dS}{p_i} \quad (1035)$$

Finally, the inverse mass tensor of the band  $i$  is computed as an averaged value over the energy space with the carrier distribution function  $f(\epsilon, F_p)$  accounted:

$$(\bar{m}_{\text{cond}, i})^{-1} = \frac{\sum_0^\infty D^i(\epsilon, z) w_{jl}^i(\epsilon, z) f(\epsilon, F_p) d\epsilon}{p_i} \quad (1036)$$

where:

- $p_i$  is a hole-band concentration ( $p_i(F_p, z) = \sum_0^\infty D^i(\epsilon, z) f(\epsilon, F_p) d\epsilon$ ).
- $(\bar{m}_{\text{cond}, i})^{-1}$  is a symmetric  $3 \times 3$  tensor, which depends on the quasi-Fermi energy  $F_p$  and the distance from interface  $z$  (for the inversion layer case).

Similar to the computation of the hole-band concentration  $p_i$ , the integral over the energy in [Equation 1036](#) is computed using Gauss–Laguerre quadratures and, for that, the energy-dependent reciprocal mass tensor ([Equation 1035](#)) is computed on a predefined energy mesh (same as for the DOS  $D^i(\epsilon, z)$ ).

Optionally, you can account for the 2D (in the vicinity of interfaces) or the 1D (in nanowires) carrier transport nature in the inverse mass tensor  $(\bar{m}_{\text{cond}, i})^{-1}$ . This is similar to recomputation of the electron transport effective masses ([Equation 1029](#) and [Equation 1030](#)). However, for holes, it is performed for all considered energies and, correspondingly, it uses the inverse mass tensor components  $w_{jl}^i(\epsilon, z)$ .

## Scattering

The scattering model defines the ratio of the stressed and unstressed valence-band momentum relaxation times  $\tau/\tau_0$ . The model considers four scattering mechanisms (which are affected by the stress) for holes in each band assisted by: acoustic phonon, optical phonon emission, optical phonon absorption, and simplified impurity scattering.

## Chapter 31: Mechanical Stress

### Mobility Modeling

The phonon-assisted relaxation times can be expressed as follows for the band  $i$  in the case of the inversion layer:

$$\begin{aligned} \frac{D^j(\varepsilon, z)}{\tau_{ac}^i(\varepsilon, z)} &= \frac{2\pi kT}{\hbar\rho} \frac{D_{ac}}{c_1}^2 \sum_j \sum_k D_{bulk}^j(\varepsilon) D_{bulk}^k(\varepsilon - \Delta\varepsilon_0^{k,i}) \\ \frac{D^j(\varepsilon, z)}{\tau_{ope}^i(\varepsilon, z)} &= \frac{\pi\hbar D_{op}^2}{\rho\varepsilon_{op}} (N_{op} + 1) \sum_j \sum_k D_{bulk}^j(\varepsilon) D_{bulk}^k(\varepsilon - \varepsilon_{op} - \Delta\varepsilon_0^{k,i}) \\ \frac{D^j(\varepsilon, z)}{\tau_{opa}^i(\varepsilon, z)} &= \frac{\pi\hbar D_{op}^2}{\rho\varepsilon_{op}} N_{op} \sum_j \sum_k D_{bulk}^j(\varepsilon) D_{bulk}^k(\varepsilon + \varepsilon_{op} - \Delta\varepsilon_0^{k,i}) \end{aligned} \quad (1037)$$

where:

- $D_{bulk}^i(\varepsilon) = \frac{2}{(2\pi)^3} \circ \frac{dS}{\varepsilon^i(\vec{k}) = \varepsilon \left| \nabla_{\vec{k}} \varepsilon^i(\vec{k}) \right|}$  is the hole bulk DOS of the band  $i$ .
- $\Delta\varepsilon_0^{k,i}$  is a band minimum energy difference between bands  $i$  and  $k$ .
- $\frac{D_{ac}}{c_1}$  is a ratio of the acoustic-phonon deformation potential by the sound velocity.
- $\rho$  is the mass density.
- $\varepsilon_{op}$  is the optical-phonon energy.
- $D_{op}$  is the optical-phonon deformation potential.
- $N_{op} = [\exp(\varepsilon_{op}/kT) - 1]^{-1}$  is the phonon number.

#### Note:

For the bulk case,  $D^j(\varepsilon, z) = D_{bulk}^j(\varepsilon)$  and, therefore, [Equation 1037](#) can be simplified to regular bulk scattering rate equations. [Equation 1037](#) accounts for both interband and intraband phonon scattering with the same deformation potentials, but you have an option to control the interband scattering factor.

Based on [Equation 1037](#), the total phonon-limited hole-scattering rate in the band  $i$  is written as:

$$\frac{1}{\tau_{ph}^i(\varepsilon, z)} = \frac{1}{\tau_{ac}^i(\varepsilon, z)} + \frac{1}{\tau_{ope}^i(\varepsilon, z)} + \frac{1}{\tau_{opa}^i(\varepsilon, z)} \quad (1038)$$

## Chapter 31: Mechanical Stress

### Mobility Modeling

Therefore, the macroscopic phonon-limited momentum relaxation time of the full valence band can be expressed similarly for electrons ([Equation 1025](#)):

$$\frac{1}{\tau_{\text{ph}}^{\text{ph}}} = \frac{i \int_0^{\infty} D^i(\epsilon, z) \frac{1}{\tau_{\text{ph}}^i(\epsilon, z)} f(\epsilon, F_p) d\epsilon}{p} \quad (1039)$$

where  $\tau^{\text{ph}}$  depends on the quasi-Fermi energy  $F_p$  and the distance from interface  $z$  (for the inversion layer case). The impurity scattering is introduced also similarly to how it is performed for electrons in [Equation 1026](#), and the ratio of the stressed and unstressed valence-band relaxation times in [Equation 1033](#) is expressed as follows:

$$\frac{\tau}{\tau_0} = \frac{1}{1 + \frac{\tau_0^{\text{ph}}}{\tau^{\text{ph}}} - 1 - \frac{1 - \beta^{-1}}{1 + \frac{N_{\text{tot}}}{N_{\text{ref}}}^{\alpha}}} \quad (1040)$$

where:

- $\tau_0^{\text{ph}}$  is the phonon-limited momentum relaxation time in the valence band for zero stress computed using [Equation 1038](#) and [Equation 1039](#).
- $N_{\text{tot}}$  is the doping concentration.
- $N_{\text{ref}}, \alpha$  are fitting parameters of the impurity scattering model.
- $\beta$  is a fitting parameter that can be represented as  $\beta = 1 + \tau^{\text{indep}} / \tau_0^{\text{ph}}$ , where  $\tau^{\text{indep}}$  is a relaxation time of other scattering mechanisms that are not accounted for in [Equation 1037](#) and that are independent of the stress. Therefore, for example, if the stress-independent scattering rate (not accounted for in [Equation 1037](#)) is much higher than  $1/\tau_0^{\text{ph}}$ , then  $\beta = 1$  and therefore  $\tau/\tau_0 = 1$ . However, if this rate is negligible, then  $\beta = 0$ .

#### Note:

The default value of the parameter  $\beta$  differs for the bulk and inversion layer simulations. Some fitting of this parameter was performed to obtain a reasonable agreement of the model ([Equation 1033](#)) to multiple stress and orientation data.

## Using Multivalley Hole Mobility Model

The stress-induced hole mobility model can be activated with `hMultivalley(kpDOS)` because the model uses the six-band k·p valence band structure. The model works in two modes: bulk and inversion layer (in the presence of `hMultivalley(MLDA kpDOS)`) and inside semiconductor–insulator interface vicinity. Generally, all valleys defined by the multivalley model (for example, with `hMultivalley(kpDOS parfile)`; see [Multivalley Band Structure on page 327](#)) will be used in the model. However, it is calibrated only for silicon, germanium, and SiGe band structures with the six-band k·p model. With

## Chapter 31: Mechanical Stress

### Mobility Modeling

`hMultivalley(MLDA)`, the multivalley MLDA quantization model (see [Modified Local-Density Approximation Model on page 370](#)) is applied to both the density and the inversion layer stress-induced mobility model. To use another quantization model in the density computation, specify the `-Density` option in the `hMultivalley` statement (see [Notes on the Use of the MLDA Model on page 377](#)).

The model can be used regionwise or materialwise and, typically, for PMOSFET stress simulations, it can be activated with the following keyword in the `Mobility` statement of the `Piezo` model:

```
Physics {
    hMultivalley( MLDA kpDOS )
    Piezo( Model(
        Mobility(hSubband(Fermi EffectiveMass Scattering(MLDA)))
    )))
}
```

The keyword `Fermi` defines that the Fermi–Dirac distribution function is used in [Equation 1036](#) and [Equation 1039](#) with the carrier self-consistent quasi-Fermi energy. To activate only doping-dependent quasi-Fermi energy, the keywords `hSubband(Doping)` should be used. If neither of these keywords (`Fermi` and `Doping`) is used, the Boltzmann distribution function is used and this illuminates the quasi-Fermi energy dependency in the mobility model ([Equation 1033](#)).

#### Note:

If the model is used without options, as `hSubband`, then the recommended options are activated, and this is equivalent to `hSubband(Doping EffectiveMass Scattering(MLDA))`.

The keyword `EffectiveMass` means that the inverse mass tensor ([Equation 1036](#)) is computed. If this keyword is not present, the unit diagonal (and stress-independent) tensor is used as  $(m_{\text{cond}, i})$  in [Equation 1033](#). Using `EffectiveMass(Transport)` activates the 2D inverse conductivity mass tensor recomputation (similar to [Equation 1029](#) for electrons). With the option `EffectiveMass(Transport<vector>)`, you can activate the 1D transport mass recomputation (similar to [Equation 1030](#) for electrons) where `<vector>` represents the direction of 1D carrier transport in the simulation coordinate system.

The statement `Scattering(MLDA)` means that the scattering model ([Equation 1037](#)–[Equation 1040](#)) is used. If the keyword `scattering` is not present, then  $\tau/\tau_0$  is unit and stress independent in [Equation 1033](#). The keyword `MLDA` in the `Scattering` statement defines that MLDA DOS is used in the scattering rate (exactly as it is in [Equation 1037](#)), but using only the keyword `Scattering` (without `MLDA`) gives the bulk scattering rates ( $D'(\varepsilon, z) = D'_{\text{bulk}}(\varepsilon)$  in [Equation 1037](#)).

## Chapter 31: Mechanical Stress

### Mobility Modeling

The scattering model parameters (see [Equation 1037](#) and [Equation 1040](#)) can be specified in the parameter file in the `StressMobility` section as follows:

```
StressMobility {  
    Ephonon_h = 0.0612      # [eV]  
    Dop_h = 7.47e8          # [eV/cm]  
    Dac_cl_h = 7.5e-6      # [eVs/cm]  
    beta_h = 1e10           # [1]  
    beta_mlda_h = (6.5,1.2,2.5) # [1]  
    Nref_h = 3e19           # [cm^-3]  
    alpha_h = 0.85          # [1]  
}
```

where `Ephonon_h` corresponds to  $\varepsilon_{op}$  used in [Equation 1037](#), `Dop_h` is  $D_{op}$  and `Dac_cl_h` is  $D_{ac}/c_l$  in [Equation 1037](#), `beta_h` is  $\beta$  in [Equation 1040](#) for the bulk case, and `beta_mlda_h` defines  $\beta$  for the inversion layer case for three interface orientations (100), (110), (111). These  $\beta$  values are a result of the calibration of the stress effect in mobility produced by the model in comparison to measurement and other simulation data. The parameters `Nref_h` and `alpha_h` correspond to the impurity scattering parameters  $N_{ref}$ ,  $\alpha$  in [Equation 1040](#).

All conductivity mass-related parameters (used to compute  $(\bar{m}_{cond,i})^{-1}$  in [Equation 1036](#)) are defined by the six-band  $k \cdot p$  model and must be specified in the `LatticeParameters` section (see [Using Strained Effective Masses and DOS on page 949](#)).

#### Note:

By default, the unstressed mobility  $\mu_{p0}$  (in [Equation 1020](#)) is always computed for a <110> channel orientation and for substrate orientation, which corresponds to the auto-orientation Lombardi option (see [Auto-Orientation for the Lombardi Model on page 407](#)). Such a default requires you to have calibrated the Lombardi model parameters for the <110> channel (critical for (110) interface orientation where the unstressed mobility is anisotropic). However, if you want to use <100>/ (110) Lombardi model parameters, the keyword `-RelChDir110` must be used inside the `hSubband` statement.

For a general case of the tensor reference mobility ( $\bar{\mu}_{p0}$ ), the keyword `-AutoOrientation` must be used inside the `hSubband` statement. This option can be time consuming because additional tensor operations and mobility computations must be performed for each mesh node.

All parameters of the `StressMobility` model can be mole fraction dependent. You can check this in the SiGe material parameter file.

---

## Multivalley Ballistic Mobility Model

To model the stress effect in ballistic transport (see [Ballistic Mobility Model on page 458](#)), mainly a stress-induced change of the transport effective mass must be accounted for, as it

## Chapter 31: Mechanical Stress

### Mobility Modeling

follows from [Equation 386](#), [Equation 389](#), and [Equation 390](#). These equations suggest that the total ballistic mobility is inversely proportional to  $\sqrt{m^*}$ , where  $m^*$  is the effective mass in the transport direction.

With the multivalley ballistic mobility model, the total transport conduction or valence effective masses are computed based on the following valley averaging:

$$\frac{1}{\sqrt{m^*}} = \frac{\frac{n_i}{\sqrt{m_i^*}}}{\bar{n}_i} \quad (1041)$$

where  $n_i$  is the carrier concentration, and  $m_i^*$  is the transport effective mass in valley  $i$ .

With [Equation 1041](#), the model parameter  $k$  (see [Table 80 on page 461](#)) is multiplied by the following factor:

$$f_t = \frac{m_{00}^*}{m^*}^{\beta_t} \quad (1042)$$

where  $m_{00}^*$  is the total transport mass for zero stress, and  $\beta_t$  is a fitting parameter with the default value of 0.5, which accounts for the theoretical square root dependence in [Equation 1041](#).

The ballistic mobility based on the kinetic velocity model (see [Kinetic Velocity Model on page 459](#)) has two parts: thermionic emission ([Equation 389](#)) and free carrier acceleration terms ([Equation 390](#)). It was found that the thermionic emission term might need an additional correction to model a difference of the stress effect in ballistic transport for the linear and saturation regimes of MOS devices. To account for that, the thermionic emission term ([Equation 389](#)) is multiplied by the following factor:

$$f_q = \frac{m_0^q}{m^q}^{\beta_q} \quad (1043)$$

where  $m_0^q$  and  $m^q$  are the total unstressed and stressed quantization masses computed from the valley quantization masses as similarly done in [Equation 1041](#), and  $\beta_q$  is a fitting parameter with the default value of 0. The factor  $f_q$  accounts for the quantization-related DOS change of confined carriers in the MOS channel with a stronger effect for the linear regime.

## Using Multivalley Ballistic Mobility Model

To activate stress-induced modeling in ballistic transport, first the model must be selected as described in [Using the Ballistic Mobility Model on page 460](#). Second, the multivalley electron or hole mobility models must be activated with the additional keyword `eSubband(BalMob)` or `hSubband(BalMob)`. For other options, see [Using Multivalley Electron Mobility Model on page 960](#) and [Using Multivalley Hole Mobility Model on page 966](#).

## Chapter 31: Mechanical Stress

### Mobility Modeling

#### Note:

To compute the transport masses  $m^*_0$  and  $m^*$  correctly in [Equation 1041](#), you must define the transport direction for the model by setting `EffectiveMass(Transport<vector>)` in either `eSubband` or `hSubband` statements.

The fitting parameters  $\beta_t$  and  $\beta_q$  can be specified for electrons and holes separately in the `StressMobility` section of the parameter file as:

```
StressMobility {  
    BalmobTpower = 0.5      # [1]  
    BalmobTpower_h = 0.5    # [1]  
    BalmobQpower = 0.0      # [1]  
    BalmobQpower_h = 0.0    # [1]  
}
```

where `BalmobTpower` corresponds to  $\beta_t$  and `BalmobQpower` corresponds to  $\beta_q$ . These parameters are mole fraction dependent to be used for SiGe and other applications.

#### Note:

For simulations where the stress effect in drift-diffusion mobility is calculated by other models (not by multivalley models), specifying the `NoStressInDDmob` option in `eSubband` or `hSubband` statements limits the stress effect to the ballistic mobility only.

---

## Multivalley Transferred Carrier Mobility Model

For GaAs and other III–V materials, a negative differential mobility can be observed for high driving fields. This effect is caused by a transfer of electrons into energetically higher satellite valleys with larger effective masses (see [Transferred Electron Model on page 442](#) and [Transferred Electron Model 2 on page 442](#)).

The carrier redistribution between lower and higher valleys is a result of the distribution function widening for carriers with high energies at high driving fields. Carrier redistribution can be modeled with the Sentaurus Device hydrodynamic model (see [Hydrodynamic Model for Temperatures on page 252](#)) combined with multivalley band-structure representation (see [Multivalley Band Structure on page 327](#)).

Multivalley mobility correction models should be set to account for carrier redistribution and, consequently, the effective mass change in carrier mobility (see [Multivalley Electron Mobility Model on page 953](#) and [Multivalley Hole Mobility Model on page 963](#)). Valley shifts due to stress and quantization might contribute to further carrier redistribution and can be also accounted for in these multivalley mobility correction models.

## Using Multivalley Transferred Carrier Mobility Model

This multivalley model computes a correction to the carrier mobility as generally shown in [Equation 1020](#). Therefore, for specific III–V applications, you should set the correct mobility models and parameters (see [Introduction to Mobility Models on page 385](#)).

Multiple valley parameters for material band structures are defined in Sentaurus Device parameter files (see [Using Multivalley Band Structure on page 331](#)).

To compute the mobility correction for the transferred carrier effect, you should set eSubband and eMultivalley statements as follows (similarly, set hSubband and hMultivalley for holes):

```
Physics {  
    eMultivalley( Nonparabolicity )  
    Piezo( Model(Mobility(eSubband(Fermi EffectiveMass EqRefMob))) )  
}
```

where EqRefMob is required to activate eSubband for III–V materials. The Nonparabolicity option activates nonparabolicity in the multiple valleys that affect both carrier redistribution and effective mass. For other options, see [Using Multivalley Electron Mobility Model on page 960](#) and [Using Multivalley Hole Mobility Model on page 966](#). The model can be used regionwise or materialwise.

---

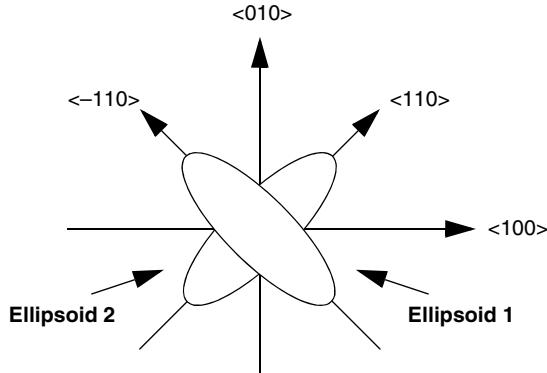
## Intel Stress-Induced Hole Mobility Model

Intel [24] suggested a mobility model for strained PMOS devices based on the occupancy of different parts of the topmost valence band. As shown in [Figure 58](#), under zero stress, the topmost valence band has a fourfold symmetry with arms along  $\langle 110 \rangle$  and  $\langle \bar{1}\bar{1}0 \rangle$ . Each ellipsoid is characterized by a transverse mass  $m_t$  and a longitudinal mass  $m_l$ .

As compressive uniaxial stress along  $\langle 110 \rangle$  is applied, one of the arms shrinks and the band becomes a single ellipsoidal band. In [Figure 58](#), this modification of the band structure is modeled as the splitting and change in curvature of two ellipsoidal bands.

With applied stress, the maximum energy associated with each of these bands splits with carriers preferentially occupying the upper valley. In addition, the transverse and longitudinal effective masses of each of these two valleys change with stress.

**Figure 58** Sketch of Intel two-ellipsoid model; one ellipsoid is aligned along  $<110>$  and the other along  $<-110>$



Under zero stress, the occupancies of the two ellipsoids are assumed equal and the mobility is isotropic with a value:

$$\mu_0 = q \tau \left[ \frac{0.5}{m_{t0}} + \frac{0.5}{m_{t0}} \right] \quad (1044)$$

where  $\tau$  is the scattering time and the index '0' in effective masses corresponds to the unstressed case.

Accounting for stress-induced reoccupation of these ellipsoids  $f_1$  and  $f_2$ , and assuming that the scattering time  $\tau$  is the same for both valleys and is independent of stress, the relative change in mobility is given by a diagonal tensor in the  $110$ ,  $\bar{1}10$  coordinate system as [24]:

$$\begin{bmatrix} 1 + \frac{\Delta\mu_{110}}{\mu_0} \\ 1 + \frac{\Delta\mu_{-110}}{\mu_0} \end{bmatrix} = \frac{2m_{t0}m_{t0}}{m_{t0} + m_{t0}} \begin{bmatrix} \frac{f_1}{m_{t1}} + \frac{f_2}{m_{t2}} & 0 \\ 0 & \frac{f_1}{m_{t1}} + \frac{f_2}{m_{t2}} \end{bmatrix} \quad (1045)$$

Denoting the energy split between the two values as  $\Delta$ , the occupancies for the two valleys are given by:

$$\begin{aligned} f_1 &= \frac{1}{1 + \exp(-\Delta/kT)} \\ f_2 &= 1 - f_1 \end{aligned} \quad (1046)$$

where  $T$  is the lattice temperature.

## Stress Dependencies

Intel decomposes the planar stress tensor in the crystallographic coordinate system as:

$$S = \begin{bmatrix} b+a & s \\ s & b-a \end{bmatrix} \quad (1047)$$

where  $b$  is the biaxial component,  $a$  is the anti-symmetric component, and  $s$  is the shear component. The basic model parameters ( $1/m_t$  and  $\Delta$ ) are expanded in powers of these components. The longitudinal mass  $m_1$  is assumed to be stress independent, that is,  $m_{11} = m_{12} = m_{10}$ .

The symmetry of the top valence band enforces some consistency relations on the basic parameters and its power-law expansions. For example, the mobility enhancement along  $110$ , when uniaxial stress along  $110$  is applied, must be equal to the enhancement along  $\bar{1}10$  when uniaxial stress along  $\bar{1}10$  is applied. In addition, when biaxial stress is applied, the mobility enhancements along  $110$  and  $\bar{1}10$  must be the same. Enforcing these symmetries, the following expressions can be written:

$$\begin{aligned} \Delta &= d_1 s \\ \frac{1}{m_{t1}} &= \frac{1}{m_{t0}}(1 + s_{t1}s + s_{t2}s^2 + b_{t1}b + b_{t2}b^2) \\ \frac{1}{m_{t2}} &= \frac{1}{m_{t0}}(1 - s_{t1}s + s_{t2}s^2 + b_{t1}b + b_{t2}b^2) \end{aligned} \quad (1048)$$

where the fitting parameters  $d_1, s_{t1}, s_{t2}, b_{t1}, b_{t2}$  can be specified in the `StressMobility` section of the parameter file. The default values of these parameters are based on fitting the model to various PMOSFET stress data described in the literature [25].

## Generalization of Model

The original Intel model considered only 2D planes. Therefore, it was extended and generalized in several issues: the three-dimensional case, doping dependence, and carrier redistribution between more than two valleys.

To generalize the model for the three-dimensional case, three  $100$  planes where the model is applied separately are considered. It is assumed that the valence band is a sum of six ellipsoids and this is consistent with the model suggested in the literature [26]. Sentaurus Device transforms the stress tensor into the crystal system and then, considering these three  $\{100\}$  planes, it selects only the corresponding components of the stress tensor. For example, for the  $[100], [010]$  plane, Sentaurus Device takes only the  $s_1, s_2$ , and  $s_6$  stress components and recomputes them into  $a, b$ , and  $s$  of [Equation 1047](#). In addition, a modification of the effective mass perpendicular to the plane (parallel to the  $[001]$  direction) was introduced to account better for  $001$  simulation cases:  $1/m_{t,001} = (1 + b_{tt}b)/m_{t0}$ , where the parameter  $b_{tt}$  can be modified in the parameter file.

## Chapter 31: Mechanical Stress

### Mobility Modeling

With this modification, the diagonal tensor of the stress-induced mobility change in the  $\bar{1}10$ ,  $\bar{1}\bar{1}0$ ,  $001$  coordinate system can be written as:

$$\begin{bmatrix} \Delta\mu_{110} \\ \Delta\mu_{-110} \\ \Delta\mu_{001} \end{bmatrix} = \mu_0 \begin{bmatrix} \frac{f_1}{m_{t1}} + \frac{f_2}{m_{t2}} / \frac{0.5}{m_{t0}} + \frac{0.5}{m_{t0}} - 1 & 0 & 0 \\ 0 & \frac{f_1}{m_{t1}} + \frac{f_2}{m_{t2}} / \frac{0.5}{m_{t0}} + \frac{0.5}{m_{t0}} - 1 & 0 \\ 0 & 0 & m_{t0}/m_{t001} - 1 \end{bmatrix} \quad (1049)$$

Next, the relative change of the mobility is computed independently for each plane and it is summed in the crystal system. Finally, the mobility change tensor is transformed from the crystal system to the simulation one.

The carrier occupation of the two valleys (see [Equation 1046](#)) is derived assuming Boltzmann statistics and the same density-of-states in both these valleys. To obtain a doping dependence, it is necessary to consider Fermi–Dirac statistics and, in this case, the following expression is obtained:

$$f_1 = \frac{1}{1 + F_{1/2} \frac{E_V - F_p - \Delta}{kT} / F_{1/2} \frac{E_V - F_p}{kT}} \quad (1050)$$

$$f_2 = 1 - f_1$$

where  $F_p$  is the hole quasi-Fermi level that is computed either assuming charge neutrality between carrier and doping, which gives only the doping dependency of the model or using carrier local concentration.

As previously discussed, the generalized model considers six ellipsoids and, therefore, the carrier reoccupation between all these valleys must be accounted for. This is not a simple problem because the stress-induced energy shift of each valley must be accounted for.

As an experimental option, Sentaurus Device gives the following simplified expressions:

$$f_1 = \frac{\frac{F_{1/2} \frac{E_V - F_p}{kT}}{(N_e - 1) F_{1/2} \frac{E_V - F_p}{kT} + F_{1/2} \frac{E_V - F_p - \Delta}{kT}}}{\frac{F_{1/2} \frac{E_V - F_p - \Delta}{kT}}{(N_e - 1) F_{1/2} \frac{E_V - F_p}{kT} + F_{1/2} \frac{E_V - F_p - \Delta}{kT}}} \quad (1051)$$

$$f_2 = \frac{\frac{F_{1/2} \frac{E_V - F_p - \Delta}{kT}}{(N_e - 1) F_{1/2} \frac{E_V - F_p}{kT} + F_{1/2} \frac{E_V - F_p - \Delta}{kT}}}{\frac{F_{1/2} \frac{E_V - F_p}{kT}}{(N_e - 1) F_{1/2} \frac{E_V - F_p}{kT} + F_{1/2} \frac{E_V - F_p - \Delta}{kT}}}$$

where  $N_e$  is equal to the number of ellipsoids that you want to consider. The value of  $N_e$  can be specified in the parameter file. The default value is 2, which transforms [Equation 1051](#) into [Equation 1050](#).

## Chapter 31: Mechanical Stress

### Mobility Modeling

#### Note:

For this multi-ellipsoid option and  $N_e > 2$ , the sum  $f_1 + f_2 < 1$  even without the stress. Therefore, [Equation 1049](#) is slightly modified also to account for this different initial occupation.

## Using Intel Mobility Model

To select the Intel stress-induced hole mobility model, you must include the following keyword in the `Mobility` statement of the `Piezo` model:

```
Physics {
    Piezo( Model(Mobility(hSixBand)) )
}
```

The keyword `hSixBand` assumes Boltzmann statistics and, in this case, [Equation 1046](#) will be activated. To have doping dependence (see [Equation 1050](#)), the keyword `hSixBand(Doping)` should be specified, but to have carrier concentration dependency (Fermi statistics), use the keywords `hSixBand(Fermi)`. The model parameters (see [Equation 1048](#)) can be specified in the `StressMobility` section of the parameter file as follows:

```
StressMobility {
    mh_10 = 0.48      # [1]
    mh_t0 = 0.15      # [1]
    ne = 2            # [1]
    d1 = -6.0000e-11  # [eV/Pa]
    st1 = -9.4426e-10 # [1/Pa]
    st2 = 4.3066e-19  # [1/Pa^2]
    bt1 = -1.0086e-10 # [1/Pa]
    bt2 = 6.5886e-21  # [1/Pa^2]
    btt = 1.2000e-10 # [1/Pa]
}
```

PMOS transistors are usually oriented in the direction to have maximum stress effect. To define this direction for the x-axis of a Sentaurus Device simulation, the following parameter set can be specified for the 2D case:

```
LatticeParameters {
    X = (1, 0, 1) #[1]
    Y = (0, 1, 0) #[1]
}
```

The `hSixBand` carrier occupancies  $f_1$  and  $f_2$  are calculated in the crystallographic coordinate system for each band.

To plot these values, use the following keywords in the `Plot` section of the command file:

- `f1BandOccupancy001` = Occupancy of the  $\bar{1}10$  ellipsoid in the (001) plane
- `f2BandOccupancy001` = Occupancy of the  $1\bar{1}0$  ellipsoid in the (001) plane

## Chapter 31: Mechanical Stress

### Mobility Modeling

- f1BandOccupancy010 = Occupancy of the 110 ellipsoid in the (010) plane
- f2BandOccupancy010 = Occupancy of the 110 ellipsoid in the (010) plane
- f1BandOccupancy100 = Occupancy of the 110 ellipsoid in the (100) plane
- f2BandOccupancy100 = Occupancy of the 110 ellipsoid in the (100) plane

## Piezoresistance Mobility Model

This approach [5][7][27] focuses on the modeling of the piezoresistive effect. The model is based on an expansion of the mobility enhancement tensor in terms of stress. Sentaurus Device provides options for computing either a first-order or a second-order piezoresistance mobility model. The first-order model accounts for a linear dependency on stress; whereas, the second-order model accounts for both linear and quadratic dependencies on stress.

The electron or hole mobility enhancement tensor, expanded up to second order in stress, is given by:

$$\frac{\mu_{ij}}{\mu_0} = \delta_{ij} + \sum_{k=1, l=1}^{3, 3} \Pi_{ijkl} \sigma_{kl} + \sum_{k=1, l=1, m=1, n=1}^{3, 3, 3, 3} \Pi_{ijklmn} \sigma_{kl} \sigma_{mn} \quad (1052)$$

where:

- $\mu_{ij}$  is a component of the electron or hole stress-dependent mobility tensor.
- $\mu_0$  denotes the isotropic mobility without stress.
- $\delta_{ij}$  is the Kronecker delta function.
- $\sigma_{kl}$  is a component of the stress tensor.
- $\Pi_{ijkl}$  is a component of the first-order electron or hole piezoconductance tensor.
- $\Pi_{ijklmn}$  is a component of the second-order electron or hole piezoconductance tensor.

The piezoconductance tensors are symmetric, which allows Equation 1052 to be written in a contracted form using index contraction (Equation 986) and conventional contraction rules (see [7], for example):

$$\frac{\mu_i}{\mu_0} = (\delta_{i1} + \delta_{i2} + \delta_{i3}) + \sum_{j=1}^6 \Pi_{ij} \sigma_j + \sum_{j=1, k=1}^6 \Pi_{ijk} \sigma_j \sigma_k \quad (1053)$$

## Chapter 31: Mechanical Stress

### Mobility Modeling

The components of the piezoconductance tensors are related to the components of the more familiar piezoresistance tensors through the following relations [7]:

$$\begin{aligned}\pi_{ij} &= -\Pi_{ij} & \leftrightarrow & \Pi_{ij} = -\pi_{ij} \\ \pi_{ijk} &= -\Pi_{ijk} + \Pi_{ij}\Pi_{ik} & \leftrightarrow & \Pi_{ijk} = -\pi_{ijk} + \pi_{ij}\pi_{ik}\end{aligned}\quad (1054)$$

where  $\pi_{ij}$  and  $\pi_{ijk}$  are components of the first-order and second-order piezoresistance tensors, respectively. When the first-order model is used (the default), only the first summation in [Equation 1053](#) is included in the calculation. When the second-order model is used, the full expression given by [Equation 1053](#) is used.

In crystals with cubic symmetry such as silicon, the number of independent coefficients of the first-order piezoresistance tensor reduces to three by rotating the coordinate system parallel to the high-symmetric axes of the crystal [8] resulting in the following  $6 \times 6$  tensor:

$$[\pi_{ij}] = \begin{bmatrix} \pi_{11} & \pi_{12} & \pi_{12} & 0 & 0 & 0 \\ \pi_{12} & \pi_{11} & \pi_{12} & 0 & 0 & 0 \\ \pi_{12} & \pi_{12} & \pi_{11} & 0 & 0 & 0 \\ 0 & 0 & 0 & \pi_{44} & 0 & 0 \\ 0 & 0 & 0 & 0 & \pi_{44} & 0 \\ 0 & 0 & 0 & 0 & 0 & \pi_{44} \end{bmatrix} \quad (1055)$$

## Chapter 31: Mechanical Stress

### Mobility Modeling

The second-order piezoresistance tensor has only nine independent components and can be expressed with the following  $6 \times 6 \times 6$  tensor:

$$\begin{aligned}
 [\pi_{1jk}] &= \begin{bmatrix} \pi_{111} & \pi_{112} & \pi_{112} & 0 & 0 & 0 \\ \pi_{112} & \pi_{122} & \pi_{123} & 0 & 0 & 0 \\ \pi_{112} & \pi_{123} & \pi_{122} & 0 & 0 & 0 \\ 0 & 0 & 0 & \pi_{144} & 0 & 0 \\ 0 & 0 & 0 & 0 & \pi_{166} & 0 \\ 0 & 0 & 0 & 0 & 0 & \pi_{166} \end{bmatrix} & [\pi_{2jk}] &= \begin{bmatrix} \pi_{122} & \pi_{112} & \pi_{123} & 0 & 0 & 0 \\ \pi_{112} & \pi_{111} & \pi_{112} & 0 & 0 & 0 \\ \pi_{123} & \pi_{112} & \pi_{122} & 0 & 0 & 0 \\ 0 & 0 & 0 & \pi_{166} & 0 & 0 \\ 0 & 0 & 0 & 0 & \pi_{144} & 0 \\ 0 & 0 & 0 & 0 & 0 & \pi_{166} \end{bmatrix} \\
 [\pi_{3jk}] &= \begin{bmatrix} \pi_{122} & \pi_{123} & \pi_{112} & 0 & 0 & 0 \\ \pi_{123} & \pi_{122} & \pi_{112} & 0 & 0 & 0 \\ \pi_{112} & \pi_{112} & \pi_{111} & 0 & 0 & 0 \\ 0 & 0 & 0 & \pi_{166} & 0 & 0 \\ 0 & 0 & 0 & 0 & \pi_{166} & 0 \\ 0 & 0 & 0 & 0 & 0 & \pi_{144} \end{bmatrix} & [\pi_{4jk}] &= \begin{bmatrix} 0 & 0 & 0 & \pi_{441} & 0 & 0 \\ 0 & 0 & 0 & \pi_{661} & 0 & 0 \\ 0 & 0 & 0 & \pi_{661} & 0 & 0 \\ \pi_{441} & \pi_{661} & \pi_{661} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \pi_{456} \\ 0 & 0 & 0 & 0 & \pi_{456} & 0 \end{bmatrix} \\
 [\pi_{5jk}] &= \begin{bmatrix} 0 & 0 & 0 & 0 & \pi_{661} & 0 \\ 0 & 0 & 0 & 0 & \pi_{441} & 0 \\ 0 & 0 & 0 & 0 & \pi_{661} & 0 \\ 0 & 0 & 0 & 0 & 0 & \pi_{456} \\ \pi_{661} & \pi_{441} & \pi_{661} & 0 & 0 & 0 \\ 0 & 0 & 0 & \pi_{456} & 0 & 0 \end{bmatrix} & [\pi_{6jk}] &= \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & \pi_{661} \\ 0 & 0 & 0 & 0 & 0 & \pi_{661} \\ 0 & 0 & 0 & 0 & 0 & \pi_{441} \\ 0 & 0 & 0 & 0 & \pi_{456} & 0 \\ 0 & 0 & 0 & \pi_{456} & 0 & 0 \\ \pi_{661} & \pi_{661} & \pi_{441} & 0 & 0 & 0 \end{bmatrix} \tag{1056}
 \end{aligned}$$

Since the coordinate system of the simulation is not necessarily parallel to the high-symmetric axis of the crystal, the orientations of the x-axis and y-axis of the crystal system can be specified in the command file (see [Using Stress and Strain on page 937](#)).

## Doping and Temperature Dependency

A simple model for the doping and temperature variation of the first-order and second-order piezoconductance coefficients is given by [7][28]:

$$\begin{aligned}
 \Pi_{ij}(N, T) &= P_1(N, T)\Pi_{ij}(0, 300\text{K}) \\
 \Pi_{ijk}(N, T) &= P_2(N, T)\Pi_{ijk}(0, 300\text{K})
 \end{aligned} \tag{1057}$$

where  $\Pi_{ij}(0, 300\text{K})$  and  $\Pi_{ijk}(0, 300\text{K})$  are piezoconductance coefficients for low-doped silicon at 300 K, and  $P_1(N, T)$  and  $P_2(N, T)$  are doping factors given by:

$$\begin{aligned}
 P_1(N, T) &= \frac{300\text{K}}{T} \frac{F_{-1}(\eta)}{F_0(\eta)} \\
 P_2(N, T) &= \frac{300\text{K}}{T}^2 \frac{F_{+2}(\eta)}{F_0(\eta)}
 \end{aligned} \tag{1058}$$

## Chapter 31: Mechanical Stress

### Mobility Modeling

In [Equation 1058](#),  $F_r(\eta)$  is a Fermi–Dirac integral of order  $r$ , and  $\eta$  is the Fermi energy measured from the band edge in units of  $kT$  ( $\eta_n = (E_{F,n} - E_C)/(kT)$  and  $\eta_p = (E_V - E_{F,p})/(kT)$ ). Doping dependency is introduced by assuming charge neutrality in the calculation of the Fermi energy.

#### Note:

For minority carriers, the doping factors  $P_1(N, T)$  and  $P_2(N, T)$  are equal to 1.

In addition, Sentaurus Device allows the calculation to be suppressed for majority carriers when the doping is below a user-specified doping threshold (see [Stress Mobility Model for Minority Carriers on page 996](#)).

The temperature and doping variation of the piezoresistance coefficients is obtained by combining [Equation 1054](#) and [Equation 1057](#). An exception to this is when the first-order model is used. In this case, the piezoresistance coefficients are split between a constant part (associated with changes in the effective masses) and a part that varies with temperature and doping (associated with anisotropic scattering):

$$\pi_{ij}(N, T) = \pi_{ij, \text{var}} P_1(N, T) + \pi_{ij, \text{con}} \quad (1059)$$

The default values of the piezoresistance coefficients for low-doped silicon at 300 K are listed in [Table 162 on page 980](#) and [Table 163 on page 980](#). They can be changed in the parameter file as described in [Using Piezoresistance Mobility Model on page 979](#).

## Using Piezoresistance Mobility Model

The command file syntax for the piezoresistance mobility models, including options, is:

```
Physics {
    Piezo (
        Model (
            Mobility (
                Tensor (
                    [FirstOrder | SecondOrder] [Enormal | <pmi_model>] [Kanda]
                    [ParameterSetName= "<psname>" | AutoOrientation]
                )
            )
        )
    )
}
```

The keyword `Tensor` selects the anisotropic tensor model for electron and hole mobility. Alternatively, the keywords `eTensor` or `hTensor` can be used to select this model for only one carrier.

The keyword `FirstOrder` or `SecondOrder` selects the first-order or second-order model, respectively. The default is `FirstOrder`. When the `FirstOrder` model is used, Sentaurus Device uses the piezoresistance coefficients shown in [Table 162](#). These can be changed from their default values in the `Piezoresistance` section of the parameter file.

## Chapter 31: Mechanical Stress

### Mobility Modeling

*Table 162 Piezoresistance coefficients for FirstOrder model: Defaults for silicon*

Symbol	Parameter name	Electrons	Holes	Unit
$\pi_{11, \text{var}}$	p11var	$-1.026 \times 10^{-9}$	$1.5 \times 10^{-11}$	$\text{Pa}^{-1}$
$\pi_{12, \text{var}}$	p12var	$5.34 \times 10^{-10}$	$1.5 \times 10^{-11}$	$\text{Pa}^{-1}$
$\pi_{44, \text{var}}$	p44var	$-1.36 \times 10^{-10}$	$1.1 \times 10^{-9}$	$\text{Pa}^{-1}$
$\pi_{11, \text{con}}$	p11con	0.0	$5.1 \times 10^{-11}$	$\text{Pa}^{-1}$
$\pi_{12, \text{con}}$	p12con	0.0	$-2.6 \times 10^{-11}$	$\text{Pa}^{-1}$
$\pi_{44, \text{con}}$	p44con	0.0	$2.8 \times 10^{-10}$	$\text{Pa}^{-1}$

When the `SecondOrder` model is used, Sentaurus Device uses the piezoresistance coefficients shown in [Table 163](#). These also can be changed from their default values in the `Piezoresistance` section of the parameter file. Note that the `SecondOrder` model uses a different set of first-order piezoresistance coefficients than the `FirstOrder` model. This allows the `FirstOrder` and `SecondOrder` models to be calibrated separately.

*Table 163 Piezoresistance coefficients for SecondOrder model: Defaults for silicon*

Symbol	Parameter name	Electrons	Holes	Unit
$\pi_{11}$	p11	$-1.1 \times 10^{-9}$	0.0	$\text{Pa}^{-1}$
$\pi_{12}$	p12	$4.5 \times 10^{-10}$	$2.0 \times 10^{-11}$	$\text{Pa}^{-1}$
$\pi_{44}$	p44	$2.5 \times 10^{-10}$	$1.19 \times 10^{-9}$	$\text{Pa}^{-1}$
$\pi_{111}$	p111	$6.6 \times 10^{-19}$	$-4.5 \times 10^{-19}$	$\text{Pa}^{-2}$
$\pi_{112}$	p112	$-5.5 \times 10^{-20}$	$2.8 \times 10^{-19}$	$\text{Pa}^{-2}$
$\pi_{122}$	p122	$-2.2 \times 10^{-20}$	$-2.5 \times 10^{-19}$	$\text{Pa}^{-2}$
$\pi_{123}$	p123	$8.8 \times 10^{-19}$	$2.0 \times 10^{-20}$	$\text{Pa}^{-2}$

*Table 163 Piezoresistance coefficients for SecondOrder model: Defaults for silicon*

Symbol	Parameter name	Electrons	Holes	Unit
$\pi_{144}$	p144	$1.0 \times 10^{-20}$	$-3.3 \times 10^{-19}$	$\text{Pa}^{-2}$
$\pi_{166}$	p166	$-6.9 \times 10^{-19}$	$6.6 \times 10^{-19}$	$\text{Pa}^{-2}$
$\pi_{661}$	p661	$6.0 \times 10^{-21}$	$-3.1 \times 10^{-19}$	$\text{Pa}^{-2}$
$\pi_{456}$	p456	$2.0 \times 10^{-20}$	$-3.0 \times 10^{-19}$	$\text{Pa}^{-2}$
$\pi_{441}$	p441	$2.0 \times 10^{-20}$	0.0	$\text{Pa}^{-2}$

The keyword `Kanda` activates the calculation of the doping factors  $P_1(N, T)$  and  $P_2(N, T)$  (see [Equation 1058](#)). If `Kanda` is not specified, these factors are equal to 1.

## Named Parameter Sets for Piezoresistance

The `Piezoresistance` parameter set can be named. For example, in the parameter file, you can write the following to declare a parameter set with the name `myset`:

```
Piezoresistance "myset" { ... }
```

To use a named parameter set, specify its name with `ParameterSetName` as an option to `Tensor` as shown in the command file syntax (see [Using Piezoresistance Mobility Model on page 979](#)).

By default, the unnamed parameter set is used.

## Auto-Orientation for Piezoresistance

The piezoresistance models support the auto-orientation framework (see [Auto-Orientation Framework on page 86](#)) that switches between different named parameter sets based on the orientation of the nearest interface. This can be activated by specifying `AutoOrientation` as an argument to `Tensor` in the command file.

## Enormal- and MoleFraction-Dependent Piezo Coefficients

Measured data shows that the piezoresistive coefficients can have dependencies with respect to the normal electric field  $E_{\perp}$  or the mole fraction  $x$  or both. Sentaurus Device has a calibration option to specify these dependencies of the piezoresistive coefficients. This model is based on the piezoresistive prefactors  $P_{ij} = P_{ij}(E_{\perp}, x)$ .

The new coefficients are calculated from:

$$\pi_{ij, \text{new}} = P_{ij}(E_{\perp}x) \cdot \pi_{ij} \quad (1060)$$

Sentaurus Device allows a piecewise linear or spline approximation (the third degree) of the prefactors over the normal electric field and a piecewise linear or piecewise cubic approximation over the mole fraction (see [Ternary Semiconductor Composition on page 74](#)).

## Using Piezoresistive Prefactors Model

To activate this model, add `Enormal` to the subsection `{e,h}Tensor`. For example:

```
Physics {
    ...
    Piezo( Model(Mobility(eTensor(Kanda Enormal))) )
}
```

The implementation of this model is based on the PMI (see [Piezoresistive Coefficients on page 1386](#)). The keyword `Enormal` means that Sentaurus Device uses the PMI predefined model `PmiEnormalPiezoResist`. You can create your own PMI models and, in this case, the keyword `Enormal` must be replaced by the name of the PMI model (for example, `eTensor("my_pmi_model")`) and the parameter file must contain a corresponding section.

**Note:**

Piezoresistive prefactors are only available when using the `FirstOrder` piezoresistance mobility model. In addition, named parameter sets and auto-orientation are not supported for piezoresistive prefactors.

By default, the values of all prefactors are equal to 1. These values can be changed in the section `PmiEnormalPiezoResist` of the parameter file. The following examples show how to use this section.

### Example 1

This example shows the section of the parameter file for purely `Enormal`-dependent prefactors:

```
Material = "Silicon" {
    * Example 1: Only Enormal dependence of piezoresistive coefficients
    PmiEnormalPiezoResist
    {
        eEnormalFormula = 2 # cubic spline
        *             = 0 # no Enormal dependence (default)
        *             = 1 # piecewise linear approximation
        *             = 2 # cubic spline approximation

        eNumberOfEnormalNodes = 3      # number of nodes for Spline(Enormal)

        # _k is _"index of node"
        eEnormal_1 = 1.0e+5      # [V/cm]
```

## Chapter 31: Mechanical Stress

### Mobility Modeling

```
eP11_1 = 1.0          # [1]
eP12_1 = 1.0          # [1]
eP44_1 = 1.0          # [1]

eEnormal_2 = 4.0e+5   # [V/cm]
eP11_2 = 0.75         # [1]
eP12_2 = 0.75         # [1]
eP44_2 = 0.75         # [1]

eEnormal_3 = 7.0e+5   # [V/cm]
eP11_3 = 0.5          # [1]
eP12_3 = 0.5          # [1]
eP44_3 = 0.5          # [1]

hEnormalFormula = 1           # piecewise linear approximation
hNumberOfEnormalNodes = 4      # number of nodes

# _k is _"index of the node"
hEnormal_1 = 1.0e+5   # [V/cm]
hP11_1 = 1.0          # [1]
hP12_1 = 1.0          # [1]
hP44_1 = 1.0          # [1]

hEnormal_2 = 1.9e+5   # [V/cm]
hP11_2 = 0.969625    # [1]
hP12_2 = 0.969625    # [1]
hP44_2 = 0.969625    # [1]

hEnormal_3 = 6.1e+5   # [V/cm]
hP11_3 = 0.530375    # [1]
hP12_3 = 0.530375    # [1]
hP44_3 = 0.530375    # [1]

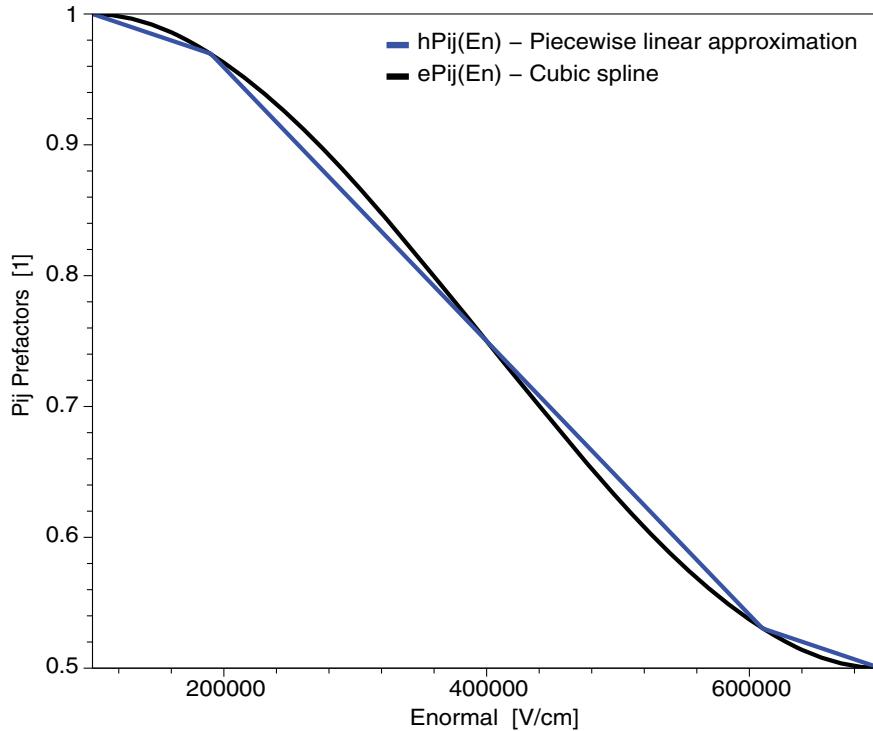
hEnormal_4 = 7.0e+5   # [V/cm]
hP11_4 = 0.5          # [1]
hP12_4 = 0.5          # [1]
hP44_4 = 0.5          # [1]
}
```

Figure 59 shows the dependency of these piezo prefactors with respect to Enormal.

## Chapter 31: Mechanical Stress

### Mobility Modeling

Figure 59 Example of cubic spline and piecewise linear approximation of piezo prefactors



### Example 2

This example shows the section of the parameter file for purely MoleFraction-dependent prefactors:

```
Material = "SiliconGermanium" {
  * Mole dependent material: SiliconGermanium (x=0) = Silicon
  * Mole dependent material: SiliconGermanium (x=1) = Germanium
  * Example 2: Only MoleFraction dependence.
  PmiEnormPiezoResist
  {
    eMoleFormula = 1                      # piecewise linear approximation
    eMoleFractionIntervals = 1            # number of MoleFraction intervals
                                         # i.e. x0=0, x1=1

    # _i is _"index of mole fraction value"
    eXmax_0 = 0
    eP11_0 = 1 # [1]
    eP12_0 = 1 # [1]
    eP44_0 = 1 # [1]

    eXmax_1 = 1
    eP11_1 = 0.5 # [1]
    eP12_1 = 0.5 # [1]
    eP44_1 = 0.5 # [1]
```

## Chapter 31: Mechanical Stress

### Mobility Modeling

```
hMoleFormula = 2      # piecewise cubic approximation
hMoleFractionIntervals = 2
    # see Ternary Semiconductor Composition on page 74
    # P = P[i-1] + A[i]*dx + B[i]*dx^2 + C[i]*dx^3
    # where:
    # A[i] = (P[i]-P[i-1])/dx[i] - B[i]*dx[i] - C[i]*dx[i]
    # dx[i] = xMax[i] - xMax[i-1]
    # dx     = x - xMax[i-1]
    # i = 1,...,nbMoleIntervals

    # _i is _"index of mole fraction value"
hXmax_0 = 0
    hP11_0 = 1 # [1]
    hP12_0 = 1 # [1]
    hP44_0 = 1 # [1]

hXmax_1 = 0.5
    hP11_1 = 2 # [1]
    hP12_1 = 2 # [1]
    hP44_1 = 2 # [1]

    # B and C coefficients
    hB11_1 = 4.
    hC11_1 = 0.
    hB12_1 = 4.
    hC12_1 = 0.
    hB44_1 = 4.
    hC44_1 = 0.

hXmax_2 = 1
    hP11_2 = 3 # [1]
    hP12_2 = 3 # [1]
    hP44_2 = 3 # [1]

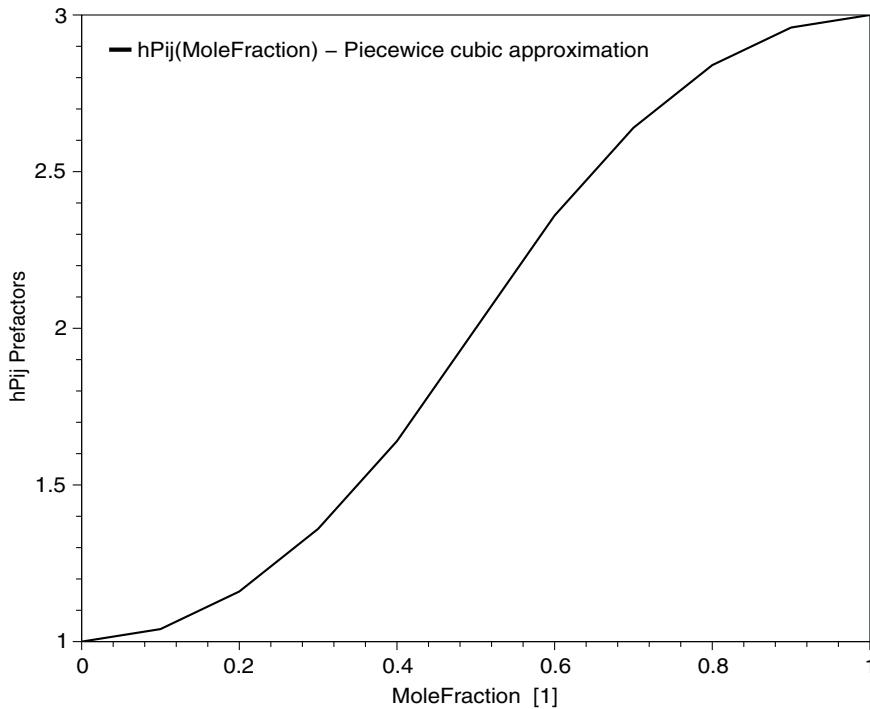
    # B and C coefficients
    hB11_2 = -4.
    hC11_2 = 0.
    hB12_2 = -4.
    hC12_2 = 0.
    hB44_2 = -4.
    hC44_2 = 0.
}
}
```

[Figure 60](#) shows the dependency of the `hPi j` prefactors of the previous example with respect to `MoleFraction`.

## Chapter 31: Mechanical Stress

### Mobility Modeling

Figure 60 Example of piecewise cubic approximation



### Example 3

This example shows the section of the parameter file for Enormal- and MoleFraction-dependent prefactors:

```
Material = "SiliconGermanium" {
  * Mole dependent material: SiliconGermanium (x=0) = Silicon
  * Mole dependent material: SiliconGermanium (x=1) = Germanium

  * Example 3: Enormal and MoleFraction dependent piezoresistance
  * prefactors (general case)
  PmiEnormPiezoResist
  {

    eMoleFormula = 2                      # piecewise cubic approximation
    eMoleFractionIntervals = 1 # i.e. x0=0, x1=1

    # begin description for mole fraction x0=0
    eXmax_0 = 0
    eEnormalFormula_0 = 2                  # cubic spline
    eNumberOfEnormalNodes_0 = 3           # number of nodes for Spline(Enormal)

    # _i_k is _"index of mole fraction value"_index of node for
    # Spline"
    eEnormal_0_1 = 1.0e+5 # [V/cm]
    eP11_0_1 = 1.0 # [1]
```

## Chapter 31: Mechanical Stress

### Mobility Modeling

```
eP12_0_1 = 1.0 # [1]
eP44_0_1 = 1.0 # [1]

eEnormal_0_2 = 4.0e+5 # [V/cm]
eP11_0_2 = 0.75 # [1]
eP12_0_2 = 0.75 # [1]
eP44_0_2 = 0.75 # [1]

eEnormal_0_3 = 7.0e+5 # [V/cm]
eP11_0_3 = 0.5 # [1]
eP12_0_3 = 0.5 # [1]
eP44_0_3 = 0.5 # [1]
# end description for mole fraction x0=0

# begin description for mole fraction x1=1
eXmax_1 = 1
eEnormalFormula_1 = 1           # piecewise linear approximation
eNumberOfEnormalNodes_1 = 2    # number of nodes

# _i_k is "index of mole fraction value"_"index of node for
# Spline"
eEnormal_1_1 = 1.0e+5 # [V/cm]
eP11_1_1 = 1.0 # [1]
eP12_1_1 = 1.0 # [1]
eP44_1_1 = 1.0 # [1]

eEnormal_1_2 = 7.0e+5 # [V/cm]
eP11_1_2 = 0.5 # [1]
eP12_1_2 = 0.5 # [1]
eP44_1_2 = 0.5 # [1]

# B and C coefficients
hB11_1 = 3.
hC11_1 = -2.
hB12_1 = 3.
hC12_1 = -2.
hB44_1 = 3.
hC44_1 = -2.

# end description for mole fraction x1=1

# hPij prefactors are equal to default values (i.e. 1)
}
```

Figure 61 shows the difference between  $I_d - V_g$  curves for a MOSFET. The parameter file section `PmiEnormPiezoResist` is the same as in [Example 1 on page 982](#). The `Physics` section is:

```
Physics {
  Fermi
  Mobility( Phumob HighFieldsat Enormal )
  EffectiveIntrinsicDensity( BandGapNarrowing(OldSlotboom) )
```

## Chapter 31: Mechanical Stress

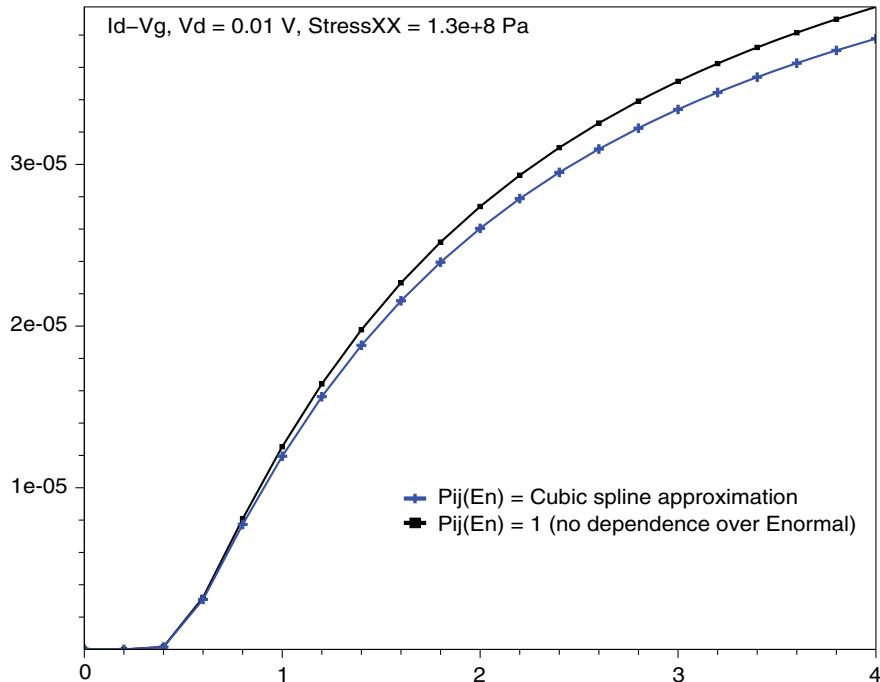
### Mobility Modeling

```

Recombination( SRH(DopingDependence) )
Piezo(
    Model( Mobility(eTensor(Kanda Enormal)) DeformationPotential )
    Stress = (1.3e8, 0, 0, 0, 0, 0)
)
}

```

Figure 61  $I_d - V_g$  curve, with *Enormal*-dependent piezo prefactors, shifts downwards




---

## Isotropic Factor Models

Sentaurus Device provides options for calculating stress-dependent enhancement factors that are applied to mobility as isotropic factors. These options, known as `Factor` models, can be used as an alternative to the tensor-based models and provide a robust approximate solution for cases where current in the device is predominantly in a direction parallel to one of the coordinate axes.

By default, the mobility enhancement factor  $\gamma$  calculated from a `Factor` model is applied to the total unstressed low-field mobility,  $\mu_{low,0}$ :

$$\mu_{low} = \gamma \mu_{low,0} \quad (1061)$$

where:

$$\gamma = 1 + \frac{\Delta\mu_{\text{low}}}{\mu_{\text{low},0}} \quad (1062)$$

However, the option `ApplyToMobilityComponents` allows the enhancement factor to be applied to individual mobility components (see [Factor Models Applied to Mobility Components on page 995](#)).

The choices for isotropic `Factor` models include `FirstOrder` or `SecondOrder` piezoresistance models, `EffectiveStressModel`, a mobility stress factor PMI model, and an `SFactor` dataset or PMI model.

## Using Isotropic Factor Models

The command file syntax for the isotropic `Factor` models, including options, is:

```
Physics {
    Piezo (
        Model (
            Mobility (
                Factor (
                    [ FirstOrder | SecondOrder |
                      EffectiveStressModel([AxisAlignedNormals]) |
                      <mobility_stress_factor_pmi_model> |
                      SFactor=<dataset_name-or-pmi_model_name>" ]
                    [ChannelDirection=<n>] [Kanda]
                    [AutoOrientation | ParameterSetName=<psname>"]
                    [ApplyToMobilityComponents]
                )
            )
        )
    )
}
```

The keyword `Factor` selects an isotropic factor model for electron and hole mobility. Alternatively, the keywords `eFactor` or `hFactor` can be used to select these models for only one carrier.

Descriptions of the model choices and options are given in the following sections.

## Piezoresistance Factor Models

These models are based on the full piezoresistance tensor calculations described in [Piezoresistance Mobility Model on page 976](#).

The piezoresistance `Factor` models use one of the diagonal components of the piezoresistance mobility enhancement tensor in the simulation coordinate system ([Equation 1053](#) after transformation from the crystallographic coordinate system) as the isotropic factor  $\gamma$  that is applied to mobility. When using this option, the channel direction

## Chapter 31: Mechanical Stress

### Mobility Modeling

must be specified using the `ChannelDirection` parameter, which determines the component of the mobility enhancement tensor to use (the default is `ChannelDirection=1`):

$$\gamma = \begin{cases} \mu'_1/\mu_0 = \mu'_{xx}/\mu_0, & \text{ChannelDirection=1} \\ \mu'_2/\mu_0 = \mu'_{yy}/\mu_0, & \text{ChannelDirection=2} \\ \mu'_3/\mu_0 = \mu'_{zz}/\mu_0, & \text{ChannelDirection=3} \end{cases} \quad (1063)$$

As with the piezoresistance tensor models, either a `FirstOrder` or `SecondOrder` model can be selected, and piezoresistance coefficients can be specified in the `Piezoresistance` section of the parameter file (see [Table 162 on page 980](#) and [Table 163 on page 980](#)).

## Effective Stress Model

The `EffectiveStressModel` is an isotropic `Factor` model that relates the mobility enhancement  $\gamma$  in a device to an effective stress parameter that is calculated from the diagonal components of the 3D stress tensor [29]:

$$\gamma = \frac{1}{\mu_0} \frac{A_1 - A_2}{1 + \exp \left[ \frac{S_{\text{eff}} - S_0}{t} \right]} + A_2 \quad (1064)$$

where:

$$A_1 = \begin{cases} \overset{\circ}{\rightarrow} a_{10} + a_{11} \frac{F_{\perp}}{10^6 \text{ V/cm}} + a_{12} \frac{F_{\perp}^2}{10^6 \text{ V/cm}^2}, & F_{\perp} \leq F_0 \times 10^6 \text{ V/cm} \\ \overset{\circ}{\rightarrow} a_{10} + a_{11}F_0 + a_{12}F_0^2, & F_{\perp} > F_0 \times 10^6 \text{ V/cm} \end{cases} \quad (1065)$$

$$A_2 = \begin{cases} \overset{\circ}{\rightarrow} a_{20} + a_{21} \frac{F_{\perp}}{10^6 \text{ V/cm}} + a_{22} \frac{F_{\perp}^2}{10^6 \text{ V/cm}^2}, & F_{\perp} \leq F_0 \times 10^6 \text{ V/cm} \\ \overset{\circ}{\rightarrow} a_{20} + a_{21}F_0 + a_{22}F_0^2, & F_{\perp} > F_0 \times 10^6 \text{ V/cm} \end{cases} \quad (1066)$$

$$S_0 = \begin{cases} \overset{\circ}{\rightarrow} s_{00} + s_{01} \frac{F_{\perp}}{10^6 \text{ V/cm}} + s_{02} \frac{F_{\perp}^2}{10^6 \text{ V/cm}^2}, & F_{\perp} \leq F_0 \times 10^6 \text{ V/cm} \\ \overset{\circ}{\rightarrow} s_{00} + s_{01}F_0 + s_{02}F_0^2, & F_{\perp} > F_0 \times 10^6 \text{ V/cm} \end{cases} \quad (1067)$$

$$t = t_0 + t_1 \frac{F_{\perp}}{10^6 \text{ V/cm}} + t_2 \frac{F_{\perp}^2}{10^6 \text{ V/cm}^2} \quad (1068)$$

## Chapter 31: Mechanical Stress

### Mobility Modeling

$F_{\perp}$  is the normal electric field and  $\mu_0$ ,  $a_{10}$ ,  $a_{11}$ ,  $a_{12}$ ,  $a_{20}$ ,  $a_{21}$ ,  $a_{22}$ ,  $s_{00}$ ,  $s_{01}$ ,  $s_{02}$ ,  $t_0$ ,  $t_1$ ,  $t_2$ , and  $F_0$  are model parameters.

### Effective Stress

In [Equation 1064](#),  $S_{\text{eff}}$  is an effective stress parameter and is given by:

$$S_{\text{eff}}(\text{MPa}) = \sum_{i=1}^3 \alpha_i \frac{S_{ii}}{10^6 \text{ Pa}} + \sum_{i=1, j \geq i}^3 \beta_{ij} \frac{S_{ii}}{10^6 \text{ Pa}} \frac{S_{jj}}{10^6 \text{ Pa}} \quad (1069)$$

where  $\alpha_i$  and  $\beta_{ij}$  are model parameters, and  $S_{ii}$  is a diagonal component of the stress tensor. The subscripts in [Equation 1069](#) have the following meaning:

$$\begin{array}{ll} \circ & 1 \rightarrow \text{channel direction} \\ i, j = & \circ 2 \rightarrow \text{nearest interface normal direction} \\ & 3 \rightarrow \text{in-plane direction} \end{array} \quad (1070)$$

The assignment of  $S_{11}$ ,  $S_{22}$ , and  $S_{33}$  is on a vertex-by-vertex basis.

$S_{11}$  is always assigned the diagonal component of stress that is associated with the `ChannelDirection` specification:

$$S_{11} = \begin{array}{ll} \sigma_{xx}, & \text{ChannelDirection}=1 \\ \sigma_{yy}, & \text{ChannelDirection}=2 \\ \sigma_{zz}, & \text{ChannelDirection}=3 \end{array} \quad (1071)$$

The assignment of  $S_{22}$  and  $S_{33}$  is performed automatically by Sentaurus Device and depends on the dimensionality of the structure, `ChannelDirection`, and the nearest interface normal direction.

In 2D structures with  $S_{11} \neq \sigma_{zz}$ :

- If  $S_{11} = \sigma_{xx}$ , then  $S_{22} = \sigma_{yy}$  and  $S_{33} = \sigma_{zz}$ .
- If  $S_{11} = \sigma_{yy}$ , then  $S_{22} = \sigma_{xx}$  and  $S_{33} = \sigma_{zz}$ .

In 3D structures, or 2D structures with  $S_{11} = \sigma_{zz}$ , the nearest interface normal might not be aligned with an axis direction. In this case, a stress transformation is made to a coordinate system ( $x'$ ,  $y'$ ,  $z'$ ) where  $x'$  is aligned with the channel direction,  $y'$  is aligned with the component of the nearest interface normal that is not in the channel direction, and  $z'$  is aligned with  $x' \times y'$ .

## Chapter 31: Mechanical Stress

### Mobility Modeling

After the stress transformation, the stress assignments are:

$$\begin{aligned} S_{11} &= \sigma_{x'x'} \\ S_{22} &= \sigma_{y'y'} \\ S_{33} &= \sigma_{z'z'} \end{aligned} \quad (1072)$$

Alternatively, the  $S_{22}$  assignment also can be made by choosing the diagonal stress component corresponding to the direction for which the nearest interface normal has its largest component (excluding any component in the channel direction). In this case, the assignment is performed as if the interface normal is aligned along an axis.  $S_{33}$  is then taken as the component that is not assigned to  $S_{11}$  or  $S_{22}$ . To select this option, specify AxisAlignedNormals as an argument to the EffectiveStressModel.

### Effective Stress Model Parameters

The EffectiveStressModel parameters can be specified in the EffectiveStressModel parameter set in the parameter file. [Table 164](#) lists the model parameters from [29] for two different surface orientations. The (100) parameters are used by default.

*Table 164 Effective stress model parameters*

Symbol	Parameter name	(100)/<110> Electrons	(100)/<110> Holes	(110)/<110> Electrons	(110)/<110> Holes	Unit
$\alpha_1$	alpha1	1.0	1.0	1.0	1.0	MPa
$\alpha_2$	alpha2	-1.7	-0.4	0.8	-0.3	MPa
$\alpha_3$	alpha3	0.7	-0.6	-1.8	-0.7	MPa
$\beta_{11}$	beta11	0.0	0.0	0.0	0.0	MPa
$\beta_{12}$	beta12	0.0	0.0	0.0	0.0	MPa
$\beta_{13}$	beta13	0.0	-0.00004	0.0	0.0	MPa
$\beta_{22}$	beta22	0.0	0.00006	0.0	0.0001	MPa
$\beta_{23}$	beta23	0.0	-0.00018	0.0	0.0	MPa
$\beta_{33}$	beta33	0.0	0.00011	0.0	0.0	MPa
$\mu_0$	mu0	810.0	212.0	326.0	1235.0	cm <sup>2</sup> /(Vs)

## Chapter 31: Mechanical Stress

### Mobility Modeling

Table 164 Effective stress model parameters (Continued)

Symbol	Parameter name	(100)/<110> Electrons	(100)/<110> Holes	(110)/<110> Electrons	(110)/<110> Holes	Unit
$a_{10}$	$a_{10}$	565.0	2460.0	270.0	505.0	$\text{cm}^2/(\text{Vs})$
$a_{11}$	$a_{11}$	-81.0	0.0	0.0	-365.0	$\text{cm}^2/(\text{Vs})$
$a_{12}$	$a_{12}$	-44.0	0.0	0.0	164.0	$\text{cm}^2/(\text{Vs})$
$a_{20}$	$a_{20}$	2028.0	42.0	761.0	9136.0	$\text{cm}^2/(\text{Vs})$
$a_{21}$	$a_{21}$	-1992.0	0.0	0.0	-25027.0	$\text{cm}^2/(\text{Vs})$
$a_{22}$	$a_{22}$	920.0	0.0	0.0	24494.0	$\text{cm}^2/(\text{Vs})$
$s_{00}$	$s_{00}$	1334.0	-1338.0	799.0	-2084.0	MPa
$s_{01}$	$s_{01}$	-2646.0	0.0	0.0	6879.0	MPa
$s_{02}$	$s_{02}$	875.0	0.0	0.0	-6896.0	MPa
$t_0$	$t_0$	882.0	524.0	417.0	-650.0	MPa
$t_1$	$t_1$	-987.0	0.0	0.0	0.0	MPa
$t_2$	$t_2$	604.0	0.0	0.0	0.0	MPa
$F_0$	$F_0$	$10^{10}$	$10^{10}$	$10^{10}$	0.5	1
$F_{\text{fixed}}$	$F_{\text{fixed}}$	-1	-1	-1	-1	MV/cm

By default, the  $F_{\text{fixed}}$  parameter is ignored. Specifying  $F_{\text{fixed}} \geq 0$  causes a fixed value of the normal field to be used in the evaluation of  $\gamma$  instead of the actual normal field. This is useful for calibration or examination of model behavior.

## Mobility Stress Factor PMI Model

A mobility stress factor PMI model (see [Mobility Stress Factor on page 1377](#)) created by the user can be utilized to obtain a mobility enhancement factor. In addition to stress, this type of PMI model allows a dependency on the normal electric field.

If specified, the Factor options ChannelDirection=<n>, AutoOrientation, and ParameterSetName="<psname>" are passed as parameters to the mobility stress factor PMI model (AutoOrientation is passed as AutoOrientation=1).

## SFactor Dataset or PMI Model

The SFactor parameter can be used to obtain an isotropic mobility enhancement factor from a dataset or a PMI model.

If SFactor="<dataset\_name>" is used, the vertex values of  $\gamma$  are taken directly from the specified dataset name, which can include the PMI user fields PMIUserField0 through PMIUserField299.

If SFactor="<pmi\_model\_name>" is used, the vertex values of  $\gamma$  are calculated from a space factor PMI model (see [Space Factor on page 1479](#)). If specified, the Factor options ChannelDirection=<n>, AutoOrientation, and ParameterSetName="<psname>" are passed as parameters to the space factor PMI model (AutoOrientation is passed as AutoOrientation=1).

## Isotropic Factor Model Options

This section discusses options of the isotropic factor model.

### Kanda Parameter

The Kanda parameter can be specified for all isotropic factor models to include a doping and temperature dependency in the enhancement factor calculation. In the case of the FirstOrder and SecondOrder piezoresistance models, this is included as described in [Doping and Temperature Dependency on page 978](#). For the EffectiveStressModel and SFactor models, the enhancement factor given by [Equation 1062](#) is modified to include the  $P_1(N, T)$  factor (see [Equation 1058](#)):

$$\gamma = 1 + P_1(N, T) \frac{\Delta\mu_{\text{low}}}{\mu_{\text{low},0}} \quad (1073)$$

### Named Parameter Sets and Auto-Orientation

The piezoresistance models and the EffectiveStressModel support the use of named parameter sets (see [Named Parameter Sets on page 85](#)) and the auto-orientation framework (see [Auto-Orientation Framework on page 86](#)). To use one of these capabilities, specify ParameterSetName="<psname>" or AutoOrientation as an argument to Factor in the command file.

## Factor Models Applied to Mobility Components

By default, the isotropic mobility enhancement factor  $\gamma$  calculated by a Factor model is applied to total low-field mobility. However, the calculated enhancement factor can be applied to select mobility components (for example, only to acoustic phonon mobility or surface roughness mobility in the Lombardi model) by specifying the `ApplyToMobilityComponents` option. When this option is selected, mobility models that support this feature apply an enhancement factor  $\gamma_i$  to the  $i^{\text{th}}$  mobility component for which a stress scaling factor  $a_i$  is available:

$$\gamma_i = 1 + a_i(\gamma - 1) \quad (1074)$$

If  $a_i = 1$ , then  $\gamma_i = \gamma$ .

If  $a_i = 0$ , then  $\gamma_i = 1$  and no stress enhancement factor is applied to this component. In most cases, the stress scaling factors have a value of 0 or 1, but intermediate values are allowed.

The  $a_i$  parameters are specified in the parameter file in the parameter set associated with the mobility model. The mobility models that support this feature and the available stress scaling parameters are shown in [Table 165](#).

*Table 165 Mobility models that support the `ApplyToMobilityComponents` feature*

Model	Parameter set name	Available stress scaling parameters
Coulomb2D	Coulomb2DMobility	$a_C$
IALMob	IALMob	$a_{\text{ph},2D}$ , $a_{\text{ph},3D}$ , $a_{C,2D}$ , $a_{C,3D}$ , $a_{\text{sr}}$
Lombardi	EnormalDependence	$a_{\text{ac}}$ , $a_{\text{sr}}$
NegInterfaceCharge	NegInterfaceChargeMobility	$a_C$
PosInterfaceCharge	PosInterfaceChargeMobility	$a_C$
RCS	RCSMobility	$a_{\text{rcs}}$
RPS	RPSMobility	$a_{\text{rps}}$
ThinLayer	ThinLayerMobility	$a_{\text{bp}}$ , $a_{\text{sp}}$ , $a_{\text{tf}}$

---

## Stress Mobility Model for Minority Carriers

Measured data shows that the stress dependency of minority carrier mobility (like the mobility of carriers in the channel of a MOSFET) is different from the stress dependency of majority carrier mobility. For example, the stress effect for minority carriers might have a dependency on electric field and, perhaps, a different (or smaller) doping dependency.

As a calibration option, Sentaurus Device provides an additional factor  $\beta$  for the stress effect applied to minority carrier mobility:

$$\bar{\mu} = \mu_0 \bar{I} + \beta \frac{\Delta \bar{\mu}_{\text{stress}}}{\mu_0} \quad (1075)$$

where  $\Delta \bar{\mu}_{\text{stress}}$  is the stress-induced change of the mobility tensor for any stress model described in this chapter. The minority carrier factor  $\beta$  can be specified (the default value is equal to 1) in the `Piezo` section of the command file using the keywords `eMinorityFactor` and `hMinorityFactor` for electrons and holes, respectively. For example:

```
Physics {
    Piezo(
        Model(Mobility(eMinorityFactor=0.5 hMinorityFactor=0.5) )
    )
}
```

In some cases, it is useful to switch off the doping dependency of the stress models and also to have the minority carrier factor  $\beta$  applied to portions of the device where the carrier is actually a majority carrier (for example, in low-doped parts of MOSFET source/drain regions). This can be achieved by specifying the parameter `DopingThreshold=Nth` as an argument to `eMinorityFactor` or `hMinorityFactor`. For example:

```
Physics {
    Piezo(Model(Mobility(
        eMinorityFactor(DopingThreshold=1e18) = 0.5
        hMinorityFactor(DopingThreshold=-1e18) = 0.5
    )))
}
```

When `DopingThreshold=Nth` is specified, the behavior of Sentaurus Device depends on the relation of  $N_{\text{th}}$  to the local net doping,  $N_D - N_A$ :

$$\begin{aligned} \mu_n: N_D - N_A < N_{\text{th}} &\rightarrow \text{eMinorityFactor} = \beta \text{ is applied, stress model doping dependency is switched off} \\ \mu_p: N_D - N_A > N_{\text{th}} &\rightarrow \text{hMinorityFactor} = \beta \text{ is applied, stress model doping dependency is switched off} \end{aligned} \quad (1076)$$

**Note:**

`DopingThreshold=0` corresponds to the regular definition of minority and majority carriers. To apply the factor  $\beta$  to a portion of the majority carrier mobility, specify  $N_{\text{th}} > 0$  for electrons and  $N_{\text{th}} < 0$  for holes.

## Chapter 31: Mechanical Stress

### Mobility Modeling

When the `DopingThreshold` condition is met (and the doping dependency of the stress models is switched off), the `Tensor(Kanda)` model behaves like `Tensor()`, the `eSubBand(Doping)` model behaves like `eSubBand()`, and the `hSixBand(Doping)` model behaves like `hSixBand()`.

Specifying `eMinorityFactor=0.5` is not the same as specifying `eMinorityFactor(ThresholdDoping=0)=0.5` because, in the latter case, the stress model doping dependency is switched off for carriers where this factor is applied.

---

## Dependency of Saturation Velocity on Stress

Equation 1075 can be rewritten in the following form (see Current Densities on page 894):

$$\bar{\mu} = \mu_0 Q \begin{bmatrix} 1+t_1 & 0 & 0 \\ 0 & 1+t_2 & 0 \\ 0 & 0 & 1+t_3 \end{bmatrix} Q^T \quad (1077)$$

where:

- $t_i$  are eigenvalues of the tensor  $\beta \frac{\Delta \bar{\mu}_{\text{stress}}}{\mu_0}$ .
- $Q$  is the orthogonal matrix from eigenvectors (main directions) of this tensor.

In high electric fields, the mobility along the main directions is proportional to the saturation velocity:  $\mu_i \sim \frac{v_{\text{sat}}}{F} (1+t_i)$ .

Therefore, the dependence of the saturation velocity on stress is similar to the mobility (with the factor  $1+t_i$ ). However, measured data shows that saturation velocity can have a different stress effect or no stress effect. Sentaurus Device has a calibration option to modify the dependency of saturation velocity on stress in the following form for the main directions:

$$v_{\text{sat}, i} = v_{\text{sat}, 0} \cdot \frac{(1 + \alpha \cdot t_i)}{(1 + t_i)} \quad (1078)$$

where  $v_{\text{sat}, 0}$  is the stress-independent saturation velocity and  $\alpha$  is a user-defined scalar factor. This factor can be specified using the keyword `SaturationFactor` in the `Piezo` section of the command file. For example:

```
Physics {
    Piezo(
        Model( Mobility(SaturationFactor = 0.5) ) )
}
```

This parameter can be specified separately for electrons and holes. For example:

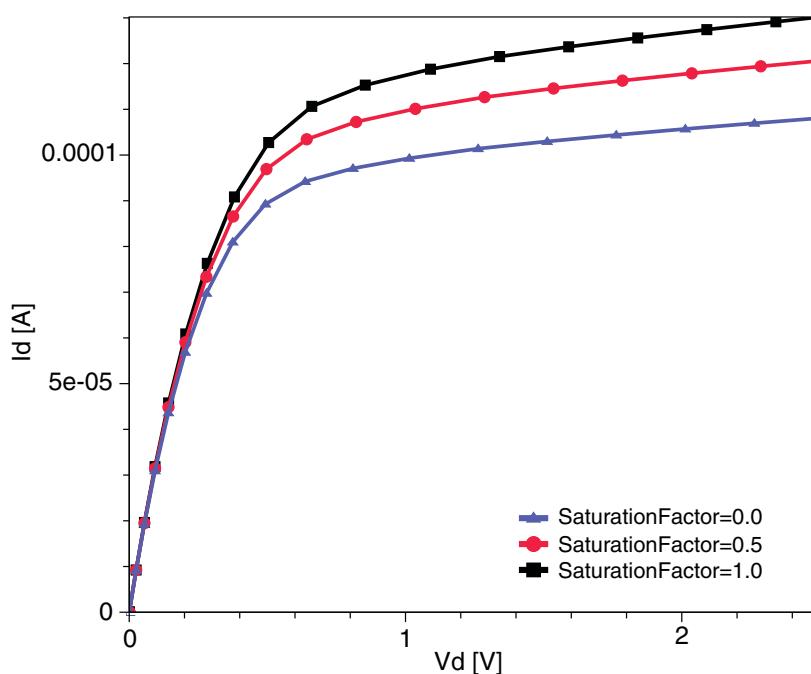
```
Physics {
    Piezo(
        Model(Mobility(eSaturationFactor=0.5 hSaturationFactor=0) ) )
```

```
}
```

The default value for  $\alpha$  is 1. This is equivalent to applying the stress enhancement factor to total mobility with  $v_{\text{sat},i} = v_{\text{sat},0}$ . Using  $\alpha = 0$  with a Caughey–Thomas-type mobility (see [Equation 354](#)) is equivalent to applying the stress enhancement factor only to the low-field mobility,  $\mu_{\text{low}}$ , with  $v_{\text{sat},i} = v_{\text{sat},0}$ .

[Figure 62](#) shows how the  $I_d$ – $V_d$  curves depend on the parameter `SaturationFactor`.

**Figure 62** Dependency of  $I_d$ – $V_d$  curves on `SaturationFactor`, with  $V_g = 1.5$  V, and  $\text{Stress} = (0.6e9, 0, 0, 0, 0, 0)$



## Mobility Enhancement Limits

With high stress values, the stress-dependent mobility models in Sentaurus Device can sometimes cause the mobility to become negative or, in some cases, to become unrealistically large. This is particularly true for the piezoresistance mobility models. To prevent this, minimum and maximum stress enhancement factors for both electron and hole mobility can be specified in the `Piezoresistance` section of the parameter file. The following example shows the default values for `MinStressFactor` and `MaxStressFactor`:

```
Piezoresistance {
    MinStressFactor = 1e-5 , 1e-5 # [1]
    MaxStressFactor = 10     , 10    # [1]
}
```

## Chapter 31: Mechanical Stress

Numeric Approximations for Tensor Mobility

### Note:

Although these parameters are specified in the `Piezoresistance` section of the parameter file, these limits are applied to all stress-dependent mobility models.

---

## Plotting Mobility Enhancement Factors

The mobility multiplication tensor ( $\bar{\mu}/\mu_0$ ) can be plotted on the mesh nodes. This tensor is a symmetric  $3 \times 3$  matrix and, therefore, has six independent values. To plot these values on the mesh nodes for electron and hole mobilities, use the keywords:

- `eMobilityStressFactorXX`, `eMobilityStressFactorYY`, `eMobilityStressFactorZZ`
- `eMobilityStressFactorYZ`, `eMobilityStressFactorXZ`, `eMobilityStressFactorXY`
- `hMobilityStressFactorXX`, `hMobilityStressFactorYY`, `hMobilityStressFactorZZ`
- `hMobilityStressFactorYZ`, `hMobilityStressFactorXZ`, `hMobilityStressFactorXY`

---

## Numeric Approximations for Tensor Mobility

This section discusses numeric approximations for tensor mobility.

---

### Tensor Grid Option

Due to an applied mechanical stress, the mobility can become a tensor. The numeric approximation of the transport equations with tensor mobility is complicated (see [Chapter 28 on page 883](#)). However, if the mesh is a tensor one, the approximation is simpler. For this option, off-diagonal mobility elements are not used and, therefore, there are no mixed derivatives in the approximation. Such an approximation gives an M-matrix property for the Jacobian and it permits stable stress simulations. Very often, critical regions are simple and the mesh constructed in such regions can be close to a tensor one.

### Note:

The off-diagonal elements of the mobility tensor appear only if there is a shear stress or the simulation coordinate system of Sentaurus Device is different from the crystal system.

By default, Sentaurus Device uses this simple tensor-grid approximation (keyword `TensorGridAniso` in the `Math` section).

The `StressSG` approximation gives the most accurate results and is independent of the mesh orientation (see [StressSG on page 885](#)). Convergence, however, might be worse than for `TensorGridAniso`.

## Chapter 31: Mechanical Stress

### Numeric Approximations for Tensor Mobility

The `AverageAniso` approximation is based on a local transformation of an anisotropic problem (stress-induced mobility tensor) to an isotropic one (see [AverageAniso on page 886](#)).

#### Note:

All approximation options do not guarantee a correct solution for arbitrary mesh and stress. Experiments with these options can give an estimation of ignored terms.

---

## Stress Tensor Applied to Low-Field Mobility

In all the previous models, the stress tensor factor was a linear factor applied to high-field mobility. Sentaurus Device allows also to apply the following diagonal mobility tensor factor to the low-field mobility:

$$\mu_{\text{high}} = \begin{bmatrix} \mu_{\text{high}}(\mu_{\text{low}} \cdot s_{xx}, \dots) & 0 & 0 \\ 0 & \mu_{\text{high}}(\mu_{\text{low}} \cdot s_{yy}, \dots) & 0 \\ 0 & 0 & \mu_{\text{high}}(\mu_{\text{low}} \cdot s_{zz}, \dots) \end{bmatrix} \quad (1079)$$

where:

- $s_{ii}$  are the diagonal elements of the stress tensor factor  $\bar{I} + \beta \frac{\Delta \bar{\mu}_{\text{stress}}}{\mu_0}$  in [Equation 1075](#). In this model, off-diagonal elements are not used.
- $\mu_{\text{high}}(\dots)$  is a scalar function that computes the high-field mobility (see [High-Field Saturation Models on page 438](#)).

This model can be specified in the `Math` section of the command file as follows:

```
Math { StressMobilityDependence = TensorFactor }
```

In this case, the dependency of saturation velocity on stress is given by:

$$v_{\text{sat}, i} = v_{\text{sat}, 0} \cdot (1 + \alpha \cdot t_i) \quad (1080)$$

which results in the same dependency on `SaturationFactor` described in [Dependency of Saturation Velocity on Stress on page 997](#) when a Caughey–Thomas-type mobility model is used.

The high-field mobility tensor (in [Equation 1079](#)) and the corresponding factors  $\mu_{\text{high}}(\mu_{\text{low}} \cdot s_{ii}, \dots) / \mu_{\text{high}}(\mu_{\text{low}}, \dots)$  can be plotted on the mesh nodes.

## Chapter 31: Mechanical Stress

### Piezoelectric Polarization

To plot these values for electron and hole mobilities, use the following keywords in the Plot section:

- eTensorMobilityXX, eTensorMobilityYY, eTensorMobilityZZ
- hTensorMobilityXX, hTensorMobilityYY, hTensorMobilityZZ
- eTensorMobilityFactorXX, eTensorMobilityFactorYY, eTensorMobilityFactorZZ
- hTensorMobilityFactorXX, hTensorMobilityFactorYY, hTensorMobilityFactorZZ

---

## Piezoelectric Polarization

Sentaurus Device provides two models (strain and stress) to compute polarization effects in GaN devices. They can be activated in the Physics section of the command file as follows:

```
Physics {
    Piezoelectric_Polarization (strain)
    Piezoelectric_Polarization (stress)
}
```

Sentaurus Device also offers a corresponding PMI model (see [Piezoelectric Polarization on page 1384](#)).

---

## Strain Model

The piezoelectric polarization vector  $P$  can be expressed as a function of the local strain tensor  $\epsilon$  as follows:

$$\begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix} = \begin{bmatrix} P_x^{\text{sp}} \\ P_y^{\text{sp}} \\ P_z^{\text{sp}} \end{bmatrix} + \begin{bmatrix} e_{11} & e_{12} & e_{13} & e_{14} & e_{15} & e_{16} \\ e_{21} & e_{22} & e_{23} & e_{24} & e_{25} & e_{26} \\ e_{31} & e_{32} & e_{33} & e_{34} & e_{35} & e_{36} \end{bmatrix} \begin{bmatrix} \epsilon_{xx} \\ \epsilon_{yy} \\ \epsilon_{zz} \\ \epsilon_{yz} \\ \epsilon_{xz} \\ \epsilon_{xy} \end{bmatrix} \quad (1081)$$

Here,  $P^{\text{sp}}$  is the spontaneous polarization vector [ $\text{C}/\text{cm}^2$ ], and the quantities  $e_{ij}$  denote the strain-charge piezoelectric coefficients [ $\text{C}/\text{cm}^2$ ].

The quantities  $P^{\text{sp}}$  and  $e_{ij}$  are defined in the crystal system. The polarization vector is first computed in crystal coordinates and is converted to simulation coordinates afterwards.

## Chapter 31: Mechanical Stress

### Piezoelectric Polarization

This general model is evaluated in either of the following circumstances:

- A constant strain tensor has been specified in the `Piezo` section of the command file.
- The strain tensor is read from a TDR file. This requires the specification of both `Strain=LoadFromFile` within the `Physics{Piezo{...}}` section and `Piezo=<file>` within the `File` section of the command file.

Otherwise, the simplified model by Ambacher described in [Simplified Strain Model](#) will be selected.

## Simplified Strain Model

This model is based on the work by Ambacher *et al.* [30][31]. It captures the first-order effect of polarization vectors in AlGaN/GaN HfETs: The interface charge induced is due to the discontinuity in the vertical component of the polarization vector at material interfaces.

The polarization vector is computed as follows:

$$\begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix} = \begin{bmatrix} P_x^{\text{sp}} \\ P_y^{\text{sp}} \\ P_z^{\text{sp}} + P_{\text{strain}} \end{bmatrix} \quad (1082)$$

where:

- $P_{\text{strain}} = 2d_{31} \cdot \text{strain} \cdot (c_{11} + c_{12} - 2c_{13}^2/c_{33})$  for  $\text{Formula}=1$ .
- $P_{\text{strain}} = 2\text{strain} \cdot (e_{31} - e_{33}c_{13}/c_{33})$  for  $\text{Formula}=2$ .
- $P^{\text{sp}}$  denotes the spontaneous polarization vector [ $\text{C}/\text{cm}^2$ ].
- $d_{31}$  is a piezoelectric coefficient [ $\text{cm}/\text{V}$ ].
- $c_{ij}$  are stiffness constants [ $\text{Pa}$ ].
- $e_{31}, e_{33}$  are strain–charge piezoelectric coefficients [ $\text{C}/\text{cm}^2$ ]. The value of strain is computed as:

$$\text{strain} = (1 - \text{relax}) \cdot (a_0 - a)/a \quad (1083)$$

where  $a_0$  represents the strained lattice constant [ $\text{\AA}$ ],  $a$  is the unstrained lattice constant [ $\text{\AA}$ ], and ‘relax’ denotes a relaxation parameter [1].

The quantities  $P^{\text{sp}}$ ,  $d_{ij}$ ,  $c_{ij}$ , and  $e_{ij}$  are defined in the crystal system. The polarization vector is first computed in crystal coordinates and is converted to simulation coordinates afterwards.

## Stress Model

Although, in most practical situations, there are only in-plane stress components due to lattice mismatch, the vertical and shear stress components give rise to in-plane piezopolarization components, which lead to volume charge densities and, therefore, potential variations.

The stress model computes the full polarization vector in tensor form without simplifying assumptions:

$$\begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix} = \begin{bmatrix} P_x^{\text{sp}} \\ P_y^{\text{sp}} \\ P_z^{\text{sp}} \end{bmatrix} + \begin{bmatrix} d_{11} & d_{12} & d_{13} & d_{14} & d_{15} & d_{16} \\ d_{21} & d_{22} & d_{23} & d_{24} & d_{25} & d_{26} \\ d_{31} & d_{32} & d_{33} & d_{34} & d_{35} & d_{36} \end{bmatrix} \begin{bmatrix} \sigma_{xx} \\ \sigma_{yy} \\ \sigma_{zz} \\ \sigma_{yz} \\ \sigma_{xz} \\ \sigma_{xy} \end{bmatrix} \quad (1084)$$

where:

- $P^{\text{sp}}$  denotes the spontaneous polarization vector [C/cm<sup>2</sup>].
- $d_{ij}$  are the piezoelectric coefficients [cm/V].

- $\begin{bmatrix} \sigma_{xx} \\ \sigma_{yy} \\ \sigma_{zz} \\ \sigma_{yz} \\ \sigma_{xz} \\ \sigma_{xy} \end{bmatrix}$  is the stress tensor [Pa].

The quantities  $P^{\text{sp}}$  and  $d_{ij}$  are defined in the crystal system. The stress tensor  $\sigma$  is defined in the stress system. It is first converted from stress coordinates to crystal coordinates. Then, the polarization vector is computed in crystal coordinates. Afterwards, the polarization vector is converted to simulation coordinates.

---

## Poisson Equation

Based on the polarization vector, the piezoelectric charge is computed according to:

$$q_{\text{PE}} = -\text{activation} \nabla P \quad (1085)$$

where **activation** is a nonnegative real calibration parameter (default is 1).

## Chapter 31: Mechanical Stress

### Piezoelectric Polarization

This value can be defined in the Physics section:

```
Physics (MaterialInterface="AlGaN/GaN") {
    Piezoelectric_Polarization (strain activation=0.5)
}
```

The  $q_{PE}$  value is added to the right-hand side of the Poisson equation:

$$\nabla \epsilon \cdot \nabla \phi = -q(p - n + N_D - N_A + q_{PE}) \quad (1086)$$

Only the first  $d$  components of the polarization vector are used to compute the polarization charge, where  $d$  denotes the dimension of the problem.

---

## Parameter File

The following parameters can be specified in the parameter file. All of them are mole fraction dependent, except `a0` and `relax`:

```
Piezoelectric_Polarization {
    # piezoelectric coefficients [cm/V]
    d11 = ...
    d12 = ...
    ...
    d36 = ...

    # spontaneous polarization [C/cm^2]
    psp_x = ...
    psp_y = ...
    psp_z = ...

    Formula = ...

    # stiffness constants [Pa]
    c11 = ...
    c12 = ...
    c13 = ...
    c33 = ...

    # piezoelectric coefficients [C/cm^2]
    e11 = ...
    e12 = ...
    ...
    e36 = ...

    # strain parameters [Å]
    a0 = ...
    a = ...
    relax = ... # [1]
}
```

---

## Coordinate Systems

The x-axis and y-axis of the simulation coordinate system are defined in the parameter file. For example:

```
LatticeParameters {  
    X = (1, 0, 0)  
    Y = (0, 0, -1)  
}
```

The z-axis is computed as the outer vector product of the x-axis and y-axis. The simulation system is defined relative to the crystal system. If the keyword `CrystalAxis` is present, then the crystal system is defined relative to the simulation system (see [Using Stress and Strain on page 937](#)).

In this example, the x-axis of the simulation system coincides with the x-axis of the crystal system. The y-axis of the simulation system runs along the negative z-axis of the crystal system. This is a common definition for 2D simulations.

If no `LatticeParameters` section is found in the parameter file, the following defaults take effect:

```
LatticeParameters {  
    X = (1, 0, 0)  
    Y = (0, 1, 0)  
}
```

The x-axis and y-axis of the stress system are defined in the `Physics` section:

```
Physics {  
    Piezo {  
        OriKddX = (-0.96 0.28 0)  
        OriKddY = (0.28 0.96 0)  
    }  
}
```

The z-axis is computed as the outer vector product of the x-axis and y-axis. The stress system is defined relative to the simulation system (see [Using Stress and Strain on page 937](#)).

---

## Converse Piezoelectric Field

The values of the converse piezoelectric field are very important for applications. You can add the dataset `ConversePiezoelectricField` to the `Plot` variables (see [Table 195 on page 1516](#)).

## Chapter 31: Mechanical Stress

### Piezoelectric Polarization

This dataset is a dimensionless tensor and is computed using the following tensor relationship:

$$\begin{bmatrix} \text{ConversePiezoelectricFieldXX} \\ \text{ConversePiezoelectricFieldYY} \\ \text{ConversePiezoelectricFieldZZ} \\ \text{ConversePiezoelectricFieldYZ} \\ \text{ConversePiezoelectricFieldXZ} \\ \text{ConversePiezoelectricFieldXY} \end{bmatrix} = \begin{bmatrix} d_{11} & d_{12} & d_{13} & d_{14} & d_{15} & d_{16} \\ d_{21} & d_{22} & d_{23} & d_{24} & d_{25} & d_{26} \\ d_{31} & d_{32} & d_{33} & d_{34} & d_{35} & d_{36} \end{bmatrix}^t \begin{bmatrix} E_x \\ E_y \\ E_z \end{bmatrix} \quad (1087)$$

where  $d_{ij}$  are piezoelectric coefficients and  $E_k$  are electric-field components.

---

## Piezoelectric Datasets

The piezoelectric polarization vector and the piezoelectric charge can be plotted by:

```
Plot {  
    PE_Polarization/vector  
    PE_Charge  
}
```

## Discontinuous Piezoelectric Charge at Heterointerfaces

Any interface has two sides (side1 and side2). If a vertex lies at the interface and heteromodels are switched on, this vertex is a double point and the piezoelectric charge has two values ( $q_1$  and  $q_2$ ). If the polarization vector is a constant vector, then  $q_1 = -q_2$  and the total charge is equal to zero. In the case of double points, Sentaurus Device creates an output file that contains two values of the piezoelectric charge, that is, the charge distributions are interface discontinuous even in homogeneous structures. Sentaurus Device contains two datasets (continuous and discontinuous) for the piezoelectric charge:

```
Plot {  
    PE_Charge    # continuous distribution in any case  
    PiezoCharge  # discontinuous distribution at the interface for  
                  # heteromodels  
}
```

---

## Gate-Dependent Polarization in GaN Devices

In this model, the polarization vector  $P$  has an additional term along the z-axis in the crystal system, which is dependent on the  $E_z$  component of the electric field [32]:

$$P_z^{\text{new}} = P_z + (e_{33}^2/c_{33}) \cdot E_z \quad (1088)$$

## Chapter 31: Mechanical Stress

### Piezoelectric Polarization

This problem can be converted into an equivalent problem with an anisotropic permittivity tensor (z-component increased by  $e_{33}^2/c_{33}$ ):

$$\nabla \bullet -\begin{bmatrix} \kappa_a E_x \\ \kappa_a E_y \\ \kappa_c E_z \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ \frac{e_{33}^2}{c_{33}} E_z \end{bmatrix} = \nabla \bullet \begin{bmatrix} \kappa_a & & \\ & \kappa_a & \\ & & \kappa_c + \frac{e_{33}^2}{c_{33}} \end{bmatrix} \nabla \phi = -\rho + \nabla \bullet P \quad (1089)$$

where  $P$  is equal to [Equation 1082](#) ( $P_{\text{strain}}$  is Formula 2).

The gate-dependent polarization model can be activated in the `Physics` section as follows:

```
Physics {
    Piezoelectric_Polarization (strain(GateDependent))
}
```

## Two-Dimensional Simulations

For a 2D simulation, you must ensure that the anisotropic direction (defined in the crystal coordinate system) lies within the xy plane of the simulation system. You can specify explicitly the transformation from the crystal (lattice) coordinate system to the simulation (mesh) coordinate system in the `LatticeParameters` section of the parameter file (see [Coordinate Systems on page 1005](#)).

If no `LatticeParameters` section is found in the parameter file, the following defaults take effect:

```
LatticeParameters {
    X = (1, 0, 0)
    Y = (0, 1, 0)
}
```

Similarly, the default anisotropic direction in a 2D simulation is the y-axis in the crystal system, which coincides with the default y-axis in the simulation system.

However, the following values of `LatticeParameters` are selected frequently for 2D simulations:

```
LatticeParameters {
    X = (1, 0, 0)
    Y = (0, 0, -1)
}
```

In this case, the y-axis in the simulation system runs along the negative z-axis in the crystal system and, therefore, it is necessary to declare the z-axis in the crystal system as the anisotropic direction:

```
Physics {
    Aniso(direction = zAxis)
}
```

## Mechanics Solver

**Note:**

The mechanics solver in Sentaurus Device is an experimental feature, and it might be modified in future releases.

During a device simulation, the stress tensor might change as a function of the solution variables. For example:

- Different materials have different thermal expansion coefficients. This thermal mismatch leads to stress changes as a function of the lattice temperature of the device. The influence of thermomechanical stress on device performance has been studied in [33] and [34].
- The electrical degradation of GaN HEMTs is described in [35]. The authors propose the formation of defects due to excessive stress associated with the inverse piezoelectric effect. In this case, the stress tensor changes as a function of the local electric field.

Sentaurus Device provides a `Mechanics` statement in the `Solve` section to recompute the stress tensor in response to changes in bias conditions:

```
Solve {
    Mechanics

    Plugin (...)
        { Poisson Electron Hole Mechanics }

    Quasistationary (...)
        { Plugin (Iterations=0 BreakOnFailure) {
            Coupled { Poisson Electron Hole }
            Mechanics
        }
    }

    Transient (...)
        { Plugin (Iterations=0 BreakOnFailure) {
            Coupled { Poisson Electron Hole }
            Mechanics
        }
    }
}
```

During a mixed-mode simulation, the `Mechanics` statement is applied by default to all the physical devices in the circuit. However, it is also possible to apply it to selected devices only:

```
Solve {
    Mechanics                      # all devices
    "mos1".Mechanics "mos2".Mechanics # selected devices only
}
```

## Chapter 31: Mechanical Stress

Mechanics Solver

Sentaurus Device relies on Sentaurus Interconnect to update the stress tensor. A `Mechanics` statement in the `Solve` section performs the following operations:

- Sentaurus Device creates an input structure (TDR file) for Sentaurus Interconnect with the current solution variables, such as the electrostatic potential  $\phi$  or the lattice temperature  $T$ , as well as a Sentaurus Interconnect command file.
- Sentaurus Device invokes Sentaurus Interconnect.
- Sentaurus Interconnect updates the mechanical stress and produces an output TDR file.
- Sentaurus Device reads the TDR file generated by Sentaurus Interconnect and updates the stress tensor.

As a consequence of this approach, mechanical equations cannot be solved self-consistently with the device equations. Therefore, a `Mechanics` statement in the `Solve` section can only appear as an individual statement, or within a `Plugin` command. It cannot appear within a `Coupled` statement.

---

## Options for the Stress Solver

Options for the stress solver can be specified in a `Mechanics` section within the global `Physics` section:

```
Physics {
    Mechanics (
        binary = "..."
        parameter = "..."
        command = "..."
        initial_structure = "..."
    )
}
```

The following options are supported:

- `binary` is the name of the Sentaurus Interconnect binary. The default is `"sinterconnect -u"` (where the `-u` option switches off the log file). Use this option to select a particular release. For example:

```
binary = "sinterconnect -rel T-2022.03"
```

The `binary` specification also can be an option of the `Mechanics` statement in the `Solve` section:

```
Solve { Mechanics (binary = "...") }
```

## Chapter 31: Mechanical Stress

### Mechanics Solver

- parameter defines Sentaurus Interconnect parameters (`pdbSet` commands). You can either specify all the parameters directly or use the `Tcl source` statement to read another file:

```
parameter = "source mechanics.par"
```

Sentaurus Interconnect can update the stress tensor as a function of the local electric field. To enable this option, use the following command:

```
pdbSet Mechanics InversePiezoEffect 1
```

By default, this option is switched off.

The values of the piezoelectric tensor  $e$  must be specified by `pdbSetDouble` commands for each region:

```
pdbSetDouble <region> Mechanics PiezoElectricTensor<ij> <value>
```

where  $i = 1 \dots 6$ ,  $j = 1 \dots 3$ . The values must be specified in units of  $\text{dyn}/(\text{V}\text{cm})$ , which is equivalent to  $10^{-2} \text{ C}/\text{cm}^2$ .

The `parameter` specification also can be an option of the `Mechanics` statement in the `Solve` section:

```
Solve { Mechanics (parameter = "...") }
```

- command defines the `Tcl` command file for Sentaurus Interconnect. You can either specify all the commands directly or use the `Tcl source` statement to read another file:

```
command = "source mechanics.cmd"
```

Sentaurus Device defines the `Tcl` variables shown in [Table 166](#) for use in the Sentaurus Interconnect command file.

*Table 166    `Tcl` variables*

Variable	Description
<code>sdevice_load</code>	Name of the file that must be loaded at the beginning by a Sentaurus Interconnect <code>init</code> command
<code>sdevice_save</code>	Name of the file that must be saved at the end by a Sentaurus Interconnect <code>struct</code> command
<code>sdevice_load_temperature</code>	Initial average device temperature in kelvin of the device defined in <code>\$sdevice_load</code>
<code>sdevice_save_temperature</code>	Final average device temperature in kelvin of the device to be saved in <code>\$sdevice_save</code>

## Chapter 31: Mechanical Stress

### References

To compute an updated stress tensor, a short `solve` command must be executed. The following commands represent the default for command:

```
init tdr= $sdevice_load !load.commands
mode mechanics
select z= PrevLatticeTemperature name= Temperature
solve time= 1<min> t.final.profile= LatticeTemperature
struct tdr= $sdevice_save
```

The `command` specification also can be an option of the `Mechanics` statement in the `Solve` section:

```
Solve { Mechanics (command = "...") }
```

- `initial_structure` specifies the input structure.

Before calling Sentaurus Interconnect, Sentaurus Device creates an input structure that contains the current solution variables, such as the electrostatic potential  $\phi$  or the lattice temperature  $T$ . During the course of a simulation, this input structure is always based on the output structure obtained from the last call of Sentaurus Interconnect.

For the first call of Sentaurus Interconnect, the input structure is based on the TDR file specified by the `initial_structure` option. It is recommended to specify a Sentaurus Interconnect structure that still contains gas regions:

```
initial_structure = "n1_fps.tdr"
```

In this way, a remeshing in Sentaurus Interconnect can be avoided.

If this option is not specified, the initial input structure will be based on the Sentaurus Device structure specified in the `File` section.

#### Note:

It is assumed that the stress tensor in the initial structure is identical to the stress tensor in Sentaurus Device (as loaded by the `Piezo` Statement in the `File` section). If a stress tensor is missing, it will be copied from the Sentaurus Device stress tensor.

---

## References

- [1] J. Bardeen and W. Shockley, “Deformation Potentials and Mobilities in Non-Planar Crystals,” *Physical Review*, vol. 80, no. 1, pp. 72–80, 1950.
- [2] I. Goroff and L. Kleinman, “Deformation Potentials in Silicon. III. Effects of a General Strain on Conduction and Valence Levels,” *Physical Review*, vol. 132, no. 3, pp. 1080–1084, 1963.

## Chapter 31: Mechanical Stress

### References

- [3] J. J. Wortman, J. R. Hauser, and R. M. Burger, "Effect of Mechanical Stress on p-n Junction Device Characteristics," *Journal of Applied Physics*, vol. 35, no. 7, pp. 2122–2131, 1964.
- [4] P. Smeys, *Geometry and Stress Effects in Scaled Integrated Circuit Isolation Technologies*, PhD thesis, Stanford University, Stanford, CA, USA, August 1996.
- [5] M. Lades *et al.*, "Analysis of Piezoresistive Effects in Silicon Structures Using Multidimensional Process and Device Simulation," in *Simulation of Semiconductor Devices and Processes (SISDEP)*, vol. 6, Erlangen, Germany, pp. 22–25, September 1995.
- [6] J. L. Egley and D. Chidambarao, "Strain Effects on Device Characteristics: Implementation in Drift-Diffusion Simulators," *Solid-State Electronics*, vol. 36, no. 12, pp. 1653–1664, 1993.
- [7] K. Matsuda *et al.*, "Nonlinear piezoresistance effects in silicon," *Journal of Applied Physics*, vol. 73, no. 4, pp. 1838–1847, 1993.
- [8] J. F. Nye, *Physical Properties of Crystals*, Oxford: Clarendon Press, 1985.
- [9] G. L. Bir and G. E. Pikus, *Symmetry and Strain-Induced Effects in Semiconductors*, New York: John Wiley & Sons, 1974.
- [10] C. Herring and E. Vogt, "Transport and Deformation-Potential Theory for Many-Valley Semiconductors with Anisotropic Scattering," *Physical Review*, vol. 101, no. 3, pp. 944–961, 1956.
- [11] E. Ungersboeck *et al.*, "The Effect of General Strain on the Band Structure and Electron Mobility of Silicon," *IEEE Transactions on Electron Devices*, vol. 54, no. 9, pp. 2183–2190, 2007.
- [12] T. Manku and A. Nathan, "Valence energy-band structure for strained group-IV semiconductors," *Journal of Applied Physics*, vol. 73, no. 3, pp. 1205–1213, 1993.
- [13] V. Sverdlov *et al.*, "Effects of Shear Strain on the Conduction Band in Silicon: An Efficient Two-Band  $k \cdot p$  Theory," in *Proceedings of the 37th European Solid-State Device Research Conference (ESSDERC)*, Munich, Germany, pp. 386–389, September 2007.
- [14] F. L. Madarasz, J. E. Lang, and P. M. Hemeier, "Effective masses for nonparabolic bands in  $p$ -type silicon," *Journal of Applied Physics*, vol. 52, no. 7, pp. 4646–4648, 1981.
- [15] C.Y.-P. Chao and S. L. Chuang, "Spin-orbit-coupling effects on the valence-band structure of strained semiconductor quantum wells," *Physical Review B*, vol. 46, no. 7, pp. 4110–4122, 1992.
- [16] M. V. Fischetti and S. E. Laux, "Band structure, deformation potentials, and carrier mobility in strained Si, Ge, and SiGe alloys," *Journal of Applied Physics*, vol. 80, no. 4, pp. 2234–2252, 1996.

## Chapter 31: Mechanical Stress

### References

- [17] C. Hermann and C. Weisbuch, “ $\vec{k} \cdot \vec{p}$  perturbation theory in III-V compounds and alloys: a reexamination,” *Physical Review B*, vol. 15, no. 2, pp. 823–833, 1977.
- [18] V. Ariel-Altschul, E. Finkman, and G. Bahir, “Approximations for Carrier Density in Nonparabolic Semiconductors,” *IEEE Transactions on Electron Devices*, vol. 39, no. 6, pp. 1312–1316, 1992.
- [19] S. Reggiani, *Report on the Low-Field Carrier Mobility Model in MOSFETs with biaxial/uniaxial stress conditions*, Internal Report, Advanced Research Center on Electronic Systems (ARCES), University of Bologna, Bologna, Italy, 2009.
- [20] S. Dhar *et al.*, “Electron Mobility Model for Strained-Si Devices,” *IEEE Transactions on Electron Devices*, vol. 52, no. 4, pp. 527–533, 2005.
- [21] E. Ungersboeck *et al.*, “Physical Modeling of Electron Mobility Enhancement for Arbitrarily Strained Silicon,” in *11th International Workshop on Computational Electronics (IWCE)*, Vienna, Austria, pp. 141–142, May 2006.
- [22] F. Stern and W. E. Howard, “Properties of Semiconductor Surface Inversion Layers in the Electric Quantum Limit,” *Physical Review*, vol. 163, no. 3, pp. 816–835, 1967.
- [23] O. Penzin, L. Smith, and F. O. Heinz, “Low Field Mobility Model for MOSFET Stress and Surface/Channel Orientation Effects,” in *42nd IEEE Semiconductor Interface Specialists Conference (SISC)*, Arlington, VA, USA, December 2011.
- [24] B. Obradovic *et al.*, “A Physically-Based Analytic Model for Stress-Induced Hole Mobility Enhancement,” in *10th International Workshop on Computational Electronics (IWCE)*, West Lafayette, IN, USA, pp. 26–27, October 2004.
- [25] L. Smith *et al.*, “Exploring the Limits of Stress-Enhanced Hole Mobility,” *IEEE Electron Device Letters*, vol. 26, no. 9, pp. 652–654, 2005.
- [26] J. R. Watling, A. Asenov, and J. R. Barker, “Efficient Hole Transport Model in Warped Bands for Use in the Simulation of Si/SiGe MOSFETs,” in *International Workshop on Computational Electronics (IWCE)*, Osaka, Japan, pp. 96–99, October 1998.
- [27] Z. Wang, *Modélisation de la piézorésistivité du Silicium: Application à la simulation de dispositifs M.O.S.*, PhD thesis, Université des Sciences et Technologies de Lille, Lille, France, 1994.
- [28] Y. Kanda, “A Graphical Representation of the Piezoresistance Coefficients in Silicon,” *IEEE Transactions on Electron Devices*, vol. ED-29, no. 1, pp. 64–70, 1982.
- [29] A. Kumar *et al.*, “A Simple, Unified 3D Stress Model for Device Design in Stress-Enhanced Mobility Technologies,” in *International Conference on Simulation of Semiconductor Processes and Devices (SISPAD)*, Denver, CO, USA, pp. 300–303, September, 2012.
- [30] O. Ambacher *et al.*, “Two-dimensional electron gases induced by spontaneous and piezoelectric polarization charges in N- and Ga-face AlGaN/GaN heterostructures,” *Journal of Applied Physics*, vol. 85, no. 6, pp. 3222–3233, 1999.

## Chapter 31: Mechanical Stress

### References

- [31] O. Ambacher *et al.*, "Two dimensional electron gases induced by spontaneous and piezoelectric polarization in undoped and doped AlGaN/GaN heterostructures," *Journal of Applied Physics*, vol. 87, no. 1, pp. 334–344, 2000.
- [32] A. Ashok *et al.*, "Importance of the Gate-Dependent Polarization Charge on the Operation of GaN HEMTs," *IEEE Transactions on Electron Devices*, vol. 56, no. 5, pp. 998–1006, 2009.
- [33] G. Y. Huang and C. M. Tan, "Electrical–Thermal–Stress Coupled-Field Effect in SOI and Partial SOI Lateral Power Diode," *IEEE Transactions on Power Electronics*, vol. 26, no. 6, pp. 1723–1732, 2011.
- [34] C. M. Tan and G. Huang, "Comparison of SOI and Partial-SOI LDMOSFETs Using Electrical–Thermal–Stress Coupled-Field Effect," *IEEE Transactions on Electron Devices*, vol. 58, no. 10, pp. 3494–3500, 2011.
- [35] J. Joh and J. A. del Alamo, "Mechanisms for Electrical Degradation of GaN High-Electron Mobility Transistors," in *IEDM Technical Digest*, San Francisco, CA, USA, pp. 1–4, December 2006.

# 32

## Galvanic Transport

---

This chapter describes the model for carrier transport in magnetic fields.

---

### Model Description

For analysis of magnetic field effects in semiconductor devices, the transport equations governing the flow of electrons and holes in the interior of the device must be set up and solved.

To this end, the commonly used drift-diffusion-based model of the carrier current densities  $\vec{J}_n$  and  $\vec{J}_p$  must be augmented by magnetic field-dependent terms that account for the action of the transport equations governing the flow of electrons and holes in the interior of the device, the *Lorentz force* on the motion of the carriers [1][2][3]:

$$\vec{J}_\alpha = \mu_\alpha \vec{g}_\alpha + \mu_\alpha \frac{1}{1 + (\mu_\alpha^* B)^2} [\mu_\alpha^* \vec{B} \times \vec{g}_\alpha + \mu_\alpha^* \vec{B} \times (\mu_\alpha^* \vec{B} \times \vec{g}_\alpha)] \quad \text{with } \alpha = n, p \quad (1090)$$

where:

- $\vec{g}_\alpha$  is current vector without mobility (see [Current Densities on page 894](#)).
- $\mu_\alpha^*$  is the Hall mobility.
- $\vec{B}$  is the magnetic induction vector, and  $B$  is the magnitude of this vector.

The perpendicular (transverse) components of Hall and drift mobility are related by  $\mu_n^* = r_n \mu_n$  and  $\mu_p^* = r_p \mu_p$ , where  $r_n$  and  $r_p$  denote the Hall scattering factors. In the case of bulk silicon, typical values are  $r_n = 1.1$  and  $r_p = -0.7$ .

---

### Using Galvanic Transport Model

In the **Physics** section of the command file, specify the magnetic field vector using the keyword **MagneticField= (<x>, <y>, <z>)**.

## Chapter 32: Galvanic Transport

Discretization Scheme for Continuity Equations

In the following example, a field of 0.1 Tesla is applied parallel to the z-axis:

```
Physics {...  
    MagneticField = (0.0, 0.0, 0.1)  
}
```

This parameter can be ramped (see [Ramping Physical Parameter Values on page 126](#)).

---

## Discretization Scheme for Continuity Equations

Sentaurus Device uses a modified discretization scheme for continuity equations in a constant magnetic field. This scheme contains an additive term to the effective electric field in the Scharfetter–Gummel approximation, that is, in this approximation, the argument to the Bernoulli function has an additional magnetic term. This discretization scheme has good convergence and mesh stability (the new approximation does not demand a special grid).

### Note:

You cannot combine the galvanic transport model with the following models: hydrodynamic, impact ionization, aniso, piezo, and quantum modeling.

The value of the magnetic field is a critical parameter for convergence. For doping-dependent mobility, convergence is reliable up to  $B = 10$  T. For more general mobility, convergence has a limit of the magnetic field up to  $B = 1$  T.

---

## References

- [1] W. Allegretto, A. Nathan, and H. Baltes, “Numerical Analysis of Magnetic-Field-Sensitive Bipolar Devices,” *IEEE Transactions on Computer-Aided Design*, vol. 10, no. 4, pp. 501–511, 1991.
- [2] C. Riccobene *et al.*, “Operating Principle of Dual Collector Magnetotransistors Studied by Two-Dimensional Simulation,” *IEEE Transactions on Electron Devices*, vol. 41, no. 7, pp. 1136–1148, 1994.
- [3] C. Riccobene *et al.*, “First Three-Dimensional Numerical Analysis of Magnetic Vector Probe,” in *IEDM Technical Digest*, San Francisco, USA, pp. 727–730, December 1994.

# 33

## Thermal Properties

---

This chapter describes the models that are important for lattice heating and the thermodynamic model: heat capacity, thermal conductivity, and thermoelectric power.

---

### Heat Capacity

Table 167 lists the values of the heat capacity used in Sentaurus Device.

Table 167    Values of heat capacity  $c$  for various materials

Material	$c$ [J/K cm <sup>3</sup> ]	Reference
Silicon	1.63	[1]
Ceramic	2.78	[2]
SiO <sub>2</sub>	1.67	[2]
Poly Si	1.63	≈ Si

By default, or when you specify `HeatCapacity(TempDep)` in the `Physics` section, the temperature dependency of the lattice heat capacity is modeled by the empirical function:

$$c_L = cv + cv_b T + cv_c T^2 + cv_d T^3 \quad (1091)$$

The equation coefficients can be specified in the parameter file by using the syntax:

```
LatticeHeatCapacity{
    cv = 1.63          # [J/(K cm^3)]
    cv_b = 0.0000e+00  # [J/(K^2 cm^3)]
    cv_c = 0.0000e+00  # [J/(K^3 cm^3)]
    cv_d = 0.0000e+00  # [J/(K^3 cm^3)]
}
```

All these coefficients can be mole fraction-dependent for mole-dependent materials.

## Chapter 33: Thermal Properties

### Heat Capacity

To use a PMI model to compute heat capacity, specify the name of the model as a string as an option to `HeatCapacity` (see [Heat Capacity on page 1339](#)). To use a multistate configuration-dependent PMI for heat capacity, specify the model and its parameters as arguments to `PMIModel`, which, in turn, is an argument to `HeatCapacity` (see [Multistate Configuration-Dependent Heat Capacity on page 1329](#)).

To use a constant lattice heat capacity without touching the parameter file, specify `HeatCapacity(Constant)`.

---

## The pmi\_msc\_heatcapacity Model

The model can depend on state occupation probabilities of a multistate configuration (MSC). If no explicit dependency on an MSC is given, then it allows a piecewise linear (pwl) dependency on the lattice temperature, that is, it reads:

$$c_V = c_V(T) \quad (1092)$$

If the model depends on an MSC, for each MSC state, then the heat capacity can be pwl temperature-dependent. The overall heat capacity is then averaged according to:

$$c_V = \sum_i c_{V,i}(T)s_i \quad (1093)$$

where the sum is taken over all MSC states, and  $s_i$  are the state occupation probabilities.

The model is activated in the `Physics` section by:

```
HeatCapacity ( PMIModel (Name="pmi_msc_heatcapacity" MSConfig="msc0" ) )
```

*Table 168 Parameters of pmi\_msc\_heatcapacity model*

Name	Symbol	Default	Unit	Range	Description
plot	—	0	—	{0,1}	Plot parameter to screen
cv	$c_V$	0.	J/Kcm <sup>3</sup>	real	Constant value
cv_nb_Tpairs	—	0	—	>=0	Number of interpolation points
cv_Tp<int>_X	—	—	K	real	Temperature at <int>-th interpolation point
cv_Tp<int>_Y	—	—	J/Kcm <sup>3</sup>	real	Value at <int>-th interpolation point

You use the `cv` parameter to set a constant heat capacity, which is used as a global (no MSC dependency) or default state heat capacity. However, if you specify `cv_nb_Tpairs` greater

## Chapter 33: Thermal Properties

### Thermal Conductivity

than zero, the global and default state heat capacities are piecewise linear. In that case, you must specify the interpolation points as pairs of temperature and heat capacity values by using `cv_Tp<int>_X` and `cv_Tp<int>_Y`, respectively, where `<int>` ranges from zero to one less than `cv_nb_Tpairs`.

The global `cv_` parameters can be prefixed with `<state_name>_` for the named MSC states to overwrite the default state behavior.

---

## Thermal Conductivity

Sentaurus Device uses the following temperature-dependent thermal conductivity  $\kappa$  in silicon [3]:

$$\kappa(T) = \frac{1}{a + bT + cT^2} \quad (1094)$$

where  $a = 0.03 \text{ cmK}^{-1}$ ,  $b = 1.56 \times 10^{-3} \text{ cmW}^{-1}$ , and  $c = 1.65 \times 10^{-6} \text{ cmW}^{-1}\text{K}^{-1}$ . The range of validity is from 200 K to well above 600 K. [Table 169](#) lists values of the thermal conductivity for some materials.

*Table 169 Values of thermal conductivity  $\kappa$  of silicon versus temperature*

Material	$\kappa$ [W/(cm K)]	Reference
Silicon	<a href="#">Equation 1094</a>	[3]
Ceramic	0.167	[4]
$\text{SiO}_2$	0.014	[1]
Poly Si	1.5	$\approx \text{Si}$

As additional options to the standard specification of the thermal conductivity model, there are two different expressions to define either thermal resistivity  $\chi = 1/\kappa$  or thermal conductivity for any material. This is performed by using `Formula` in the parameter file or special keywords in the command file.

For `Formula=0` (thermal resistivity specification), Sentaurus Device uses:

$$\chi = 1/\kappa + 1/\kappa_b T + 1/\kappa_c T^2 \quad (1095)$$

For `Formula=1` (thermal conductivity specification), it is:

$$\kappa = \kappa + \kappa_b T + \kappa_c T^2 \quad (1096)$$

## Chapter 33: Thermal Properties

### Thermal Conductivity

#### Note:

Sentaurus Device stores six independent parameters, namely:

- $1/\kappa$ ,  $1/\kappa_b$ , and  $1/\kappa_c$  for the thermal resistivity model in [Equation 1095](#)
- $\kappa$ ,  $\kappa_b$ , and  $\kappa_c$  for the thermal conductivity model in [Equation 1096](#)

Changing  $\kappa$  does not modify  $1/\kappa$  and vice versa. The same applies to  $\kappa_b$  and  $1/\kappa_b$ , as well as  $\kappa_c$  and  $1/\kappa_c$ .

Use the following syntax in the parameter file to select the required model and to specify the coefficients:

```
Kappa{
    Formula = 0
    1/kappa = 0.03          # [K cm/W]
    1/kappa_b = 1.5600e-03  # [cm/W]
    1/kappa_c = 1.6500e-06  # [cm/(W K)]
    kappa = 1.5             # [W/(K cm)]
    kappa_b = 0.0000e+00    # [W/(K^2 cm)]
    kappa_c = 0.0000e+00    # [W/(K^3 cm))]
```

The `Physics` section of the command file provides more flexibility to switch these expressions by using the keywords:

```
ThermalConductivity(
    TempDep Conductivity # Formula = 1
    Constant Conductivity # Formula = 1 without temperature dependence
    TempDep Resistivity   # Formula = 0
    Constant Resistivity   # Formula = 0 without temperature dependence
)
```

By default, Sentaurus Device uses `Formula` specified in the parameter file. All these coefficients in the parameter file can be mole dependent for mole-dependent materials.

Furthermore, a simple PMI and a multistate configuration-dependent PMI are available to compute thermal conductivity. See [Thermal Conductivity on page 1350](#) and [Multistate Configuration-Dependent Thermal Conductivity on page 1332](#).

---

## The AllDependent Thermal Conductivity Model

In general, thermal conductivity  $\kappa$  is a function of lattice temperature  $T$ , doping density  $N$ , layer thickness  $d_{Ly}$ , and mole fraction  $x$ .

## Chapter 33: Thermal Properties

### Thermal Conductivity

Starting with the Boltzmann transport equation for phonons and with relaxation time approximations,  $\kappa$  is expressed as an integral over phonon frequency [5]:

$$\kappa = \frac{1}{3} \cdot \frac{1}{2\pi^2} \cdot n_j \cdot \sum_j \int_0^{\omega_{0,j}} \frac{\tau_{\omega,j}}{v_{\omega,j}} C(\omega) \omega^2 d\omega \quad (1097)$$

$$C(x) = kx^2 e^x (e^x - 1)^{-2} \quad (1098)$$

$$x = \frac{\hbar\omega}{kT} \quad (1099)$$

For each phonon branch  $\kappa$ ,  $\omega_{0,j}$  is the peak frequency,  $v_{0,j}$  is the sound velocity, and  $\tau_{0,j}$  is the total relaxation time. For the longitudinal mode ( $j = L$ ),  $n_L = 1$ . For the transverse mode ( $j = T$ ),  $n_T = 1$ .

## Bulk Thermal Conductivity Computation

The total relaxation time is calculated with given scattering mechanisms. For pure bulk material [5]:

$$\tau_{LN}^{-1} = B_L \omega^2 T^3 + \tau_I^{-1} + \tau_b^{-1}, \quad 0 < \omega < \omega_3 \quad (1100)$$

$$\tau_{TN}^{-1} = B_T \omega T^4 + \tau_I^{-1} + \tau_b^{-1}, \quad \omega < \omega_1 \quad (1101)$$

$$\tau_{TU}^{-1} = B_{TU} \omega^2 / \sinh x + \tau_I^{-1} + \tau_b^{-1}, \quad \omega_1 < \omega < \omega_2 \quad (1102)$$

$$\tau_I^{-1} = A_I \omega^4 \quad (1103)$$

$$\tau_b^{-1} = \frac{v_s}{FL} \quad (1104)$$

$$v_s = \left[ \frac{1}{3} (v_L^{-1} + 2v_T^{-1}) \right]^{-1} \quad (1105)$$

[Equation 1100](#), [Equation 1101](#), and [Equation 1102](#) represent the phonon relaxation time for the longitudinal and transverse branches. For the transverse branch, the normal (N) process and the Umklapp (U) process are solved separately. Frequencies and phonon velocities are extracted from the bulk phonon dispersion:

- $\omega_1$  is the turning point of the transverse branch, where the U process starts to have an effect.
- $\omega_2$  is the peak frequency of the transverse branch.
- $\omega_3$  is the peak frequency of the longitudinal branch.
- $\omega_4$  is the turning point of the longitudinal branch.
- $v_L$  is the velocity of the longitudinal mode when  $\omega < \omega_4$ .
- $v_{Lp}$  is the velocity of the longitudinal mode when  $\omega_4 < \omega < \omega_3$ .

## Chapter 33: Thermal Properties

### Thermal Conductivity

- $v_T$  is the velocity of the transverse mode when  $\omega < \omega_1$ .
- $v_{TU}$  is the velocity of the transverse mode when  $\omega_1 < \omega < \omega_2$ .
- $B_L$ ,  $B_T$ , and  $B_{TU}$  are fitting parameters.
- $\tau_I^{-1}$  and  $\tau_b^{-1}$  represent scattering due to point defects and the boundary, where:
  - $A_I$  is the scattering strength due to defects.
  - $L$  is the bulk sample dimension.

The model is activated in the command file by:

```
Physics (...) {
    ThermalConductivity ( AllDep )
}
```

### Example of Parameter File Segment

```
Kappa {
    ...
    omega1 = 2.357e13    #[s^-1]
    omega2 = 2.749e13    #[s^-1]
    ...
}
```

Table 170 Summary of parameters for bulk silicon and germanium

Name	Symbol	Silicon	Germanium	Unit
omega1	$\omega_1$	2.357e13	1.322e13	1/s
omega2	$\omega_2$	2.749e13	1.545e13	1/s
omega3	$\omega_3$	7.463e13	4.36e13	1/s
omega4	$\omega_4$	4.582e13	2.514e13	1/s
vL	$v_L$	8.48e3	4.92e3	m/s
vLp	$v_{Lp}$	4.24e3	2.46e3	m/s
vT	$v_T$	5.86e3	3.55e3	m/s
vTU	$v_{TU}$	2e3	1.3e3	m/s
BL	$B_L$	2e-24	6.9e-24	s/K <sup>3</sup>

## Chapter 33: Thermal Properties

### Thermal Conductivity

Table 170 Summary of parameters for bulk silicon and germanium (Continued)

Name	Symbol	Silicon	Germanium	Unit
BT	$B_T$	9.3e-13	1e-11	1/K <sup>4</sup>
BTU	$B_{TU}$	5.5e-18	5e-18	s
AI	$A_I$	1.32e-45	2.4e-45	s <sup>3</sup>
Lb	$L$	0.716e-2	0.24e-2	m
V_h	$V$	12.1e-6	13.6e-6	m <sup>3</sup> /mol
M_h	$M$	28	72.6	Da
R_h	$R$	146e-12	125e-12	m
epsilon	$\epsilon_s$	11.7	16.2	1
m_c	$m_e$	0.9	0.9	1
m_v	$m_h$	0.58	0.58	1
rho	$\rho$	2.329e3	5.323e3	kg/m <sup>3</sup>

#### Note:

To achieve good agreement with measurement, you need to understand the relative contribution of the three phonon branches to  $\kappa$ :  $\kappa_{TU}$  mainly determines the high-temperature range. The absolute values of  $\kappa_T$  and  $\kappa_L$  determine the peak value of  $\kappa$ , while their relative ratio  $\kappa_T/\kappa_L$  affects the slope of  $\kappa$  at low temperatures: the lower the ratio, the larger the slope. The fitting parameters  $B_L$ ,  $B_T$ , and  $B_{TU}$  are determined accordingly.

## Bulk Relaxation Time With Doping

In the case of doped samples,  $\kappa$  decreases due to phonon scattering with impurities and free carriers. The phonon-impurity scattering is treated in a similar way to point defects [6]:

$$\tau_{\text{impurity}}^{-1} = A \omega^4 \quad (1106)$$

$$A = A_I + A_{\delta M} + A_{\delta R} + A_x \quad (1107)$$

## Chapter 33: Thermal Properties

### Thermal Conductivity

$$A_{\delta M} = \frac{NV^2}{4\pi v_s^3} \frac{[M - M_{\text{doped}}]^2}{M} \quad (1108)$$

$$A_{\delta R} = \frac{2NV^2}{\pi v_s^3} Q_0^2 \gamma^2 \frac{[R - R_{\text{doped}}]^2}{R} \quad (1109)$$

$A_{\delta M}$  and  $A_{\delta R}$  model the impact of mass and radius differences between the host atom and the impurity atom with the given doping concentration  $N$ , where:

- $V$  is the crystal volume of the host atom.
- $M$  is the mass of the host atom.
- $M_{\text{doped}}$  is the mass of the impurity atom.
- $R$  is the radius of the host atom.
- $R_{\text{doped}}$  is the radius of the impurity atom.
- $Q_0$  is a constant corresponding to the impurity.
- $A_x$  is a fitting parameter.

The impact of phonon-carrier scattering is considered with two parts: the free carriers in the metallic state and the carriers bound to the impurity center. With a given carrier concentration  $n$ , the concentrations of bound carriers  $N_n$  and metallic carriers  $N_m$  are [7]:

$$N_n = n \exp(-t_c) \quad (1110)$$

$$N_m = n[1 - \exp(-t_c)] \quad (1111)$$

$$t_c = (4\pi/3)n r_c^3 \quad (1112)$$

$$r_c = (144/\pi^2)^{1/3} a_B \quad (1113)$$

where the Bohr radius  $a_B$  is:

$$a_B = \frac{4\pi\epsilon_0\hbar^2}{m_0 e^2} \frac{\epsilon_s}{\epsilon_0} \frac{m_0}{m_e} \quad (1114)$$

Here:

- $\epsilon_s$  is the dielectric constant.
- $m_e$  (or  $m_h$ ) is the electron (or hole) effective mass of the host atom.

The relaxation time related to bound carriers is [8]:

$$\tau_{\text{bound},j}^{-1} = \frac{\omega^4 [(1/3)\Xi_u]^4}{10\pi\rho^2 v_s^2} \chi^2 \left[ \frac{\omega}{v_L} \right]^{-5} \left[ \chi \left( \frac{\omega}{v_L} \right)^2 + \frac{3}{2} \left( \frac{\omega}{v_T} \right)^{-5} \chi^2 \left( \frac{\omega}{v_T} \right)^2 \right] \cdot i w_j \cdot \frac{2\Delta^2}{(\Delta^2 - \hbar^2 \omega^2)^2} (2N_n) \quad (1115)$$

$$\chi(q_j) = [1 + (1/4)a_B^2 q_j^2]^{-2} \quad (1116)$$

## Chapter 33: Thermal Properties

### Thermal Conductivity

where:

- $\Xi_u$  is the shear deformation potential.
- $\rho$  is the crystal mass density.
- $w_L$ ,  $w_T$ , and  $w_{TU}$  are parameters corresponding to each phonon mode.

For carriers in metallic states, the relaxation time is [6]:

$$\tau_{\text{free}}^{-1} = \frac{(m_e E_D)^2 \omega}{2\pi \rho \hbar^3 v_s}, q < 2k_F \quad (1117)$$

$$\tau_{\text{free}}^{-1} = \frac{b N_m (m_e E_D)^2}{\rho \hbar^3 q^5 a_B^3}, q > 2k_F \quad (1118)$$

where  $E_D$  is the deformation potential, and  $b$  is a fitting parameter.

Equation 1117 and Equation 1118 are continuous at  $q = 2k_F$ , which imposes the following relation:

$$k_F = \sqrt{\frac{b N_m \pi}{32 a_B^3}}^{1/6} \quad (1119)$$

Equation 1106, Equation 1115, Equation 1117, and Equation 1118 are added to Equation 1100, Equation 1101, and Equation 1102 to solve for  $\tau_{B,j}$ , which is the bulk relaxation time.

Table 171 Summary of parameters for phosphorus and boron in silicon bulk samples

Symbol	Name	Phosphorus	Name	Boron	Unit
$M_{\text{doped}}$	M_n	30.9	M_p	10.8	Da
$R_{\text{doped}}$	R_n	123e-12	R_p	117e-12	m
$Q_0$	Q0_n	4	Q0_p	15	1
$\Xi_u$	Eu_n	9	Eu_p	45	eV
$w_L$	wL_n	0.2	wL_p	0.02	1
$w_T$	wT_n	8	wT_p	6	1
$w_{TU}$	wTU_n	0.9e6	wTU_p	2e6	1

## Chapter 33: Thermal Properties

### Thermal Conductivity

$E_D$ ,  $b$ , and  $A_x$  are fitted to experiments for a given doping concentration and doping material.

Table 172 Summary of fitting parameters for phosphorus-doped silicon bulk samples

Symbol	Name	Doping concentration [cm <sup>-3</sup> ]						Unit
		7.5e16	2.5e17	4.7e17	1e18	2e19	1.7e20	
$E_D$	ED_n	0	0.05	0.22	0.5	1.33	2.33	eV
$b$	b	0	8	8	8	186	286	1
$A_x$	Ax	0	0	11.75	11.75	6.75	11.75	$\times 10^{-45} \text{ s}^3$

Table 173 Summary of fitting parameters for boron-doped silicon bulk samples

Symbol	Name	Doping concentration [cm <sup>-3</sup> ]			Unit
		1e13	4e15	4e16	
$E_D$	ED_p	0	0	0.5	eV
$b$	b	0	0	18	1
$A_x$	Ax	0	0	1.2	$\times 10^{-45} \text{ s}^3$

#### Note:

To achieve good agreement with measurements for various doping concentrations, it is suggested to first determine  $w_L$ ,  $w_T$ , and  $w_{TU}$  for low-doping concentration samples:  $w_{TU}$  mainly determines the high-temperature range, and  $w_L$  and  $w_T$  determine the peak value and their ratio  $w_L/w_T$  determines the slope of the low-temperature range. For moderate doping cases, free-carrier scattering starts to have an effect; therefore,  $E_D$  and  $b$  are used to further fit the low-temperature range. For high doping cases, bound-carrier scattering is insignificant; therefore,  $\kappa$  is mainly determined by  $E_D$ ,  $b$ , and  $A_x$ .

In general,  $E_D$ ,  $b$ , and  $A_x$  must be fitted for each doping concentration. To provide a good guess for their values, a numerical table can be defined in the parameter file:

```
Kappa {
    NumericalTable (
        ** N ED_n ED_p b Ax
```

## Chapter 33: Thermal Properties

### Thermal Conductivity

```

7.5e16 0 0.5 8 0
2.5e17 0.05 0.5 8 0
...
)
}

```

The numerical table defines the fitted values of `ED_n`, `ED_p`, `b`, and `Ax` for some specific doping concentrations  $N$ . For other doping concentrations, these parameters are determined by interpolating the given values in the `NumericalTable`.

## Thin-Layer Relaxation Time

For thin layers with thickness  $d_{Ly}$ , the relaxation time decreases [9]:

$$\tau_{Ly,j} = F(\delta_j, p)\tau_{B,j} \quad (1120)$$

$$F(\delta_j, p) = 1 / \left[ 1 + \frac{3}{8\delta_j}(1-p) \right] \quad (1121)$$

where  $\delta_j = r_\delta d_{Ly} / (v_j \tau_{B,j})$  and  $p = \exp -\frac{16\pi^3\eta^2}{\lambda^2}$ , and where:

- $r_\delta$  is a constant to adjust the mean free path in some situations.
- $d_{Ly}$  is the layer thickness.
- $\eta$  is a fitting parameter.
- $\lambda = 2\pi v_s / \omega$ .

The layer thickness is computed internally by setting the `LayerThickness` command in the `Physics` section (see [Extracting Layer Thickness on page 379](#)). Alternatively, you can define the layer thickness by setting the parameter  $d_{Ly}$  if `LayerThickness` is not defined. By default,  $d_{Ly}$  is 0.3 m, which ensures  $F(\delta_j, p) = 1$ , thereby the bulk relaxation time is taken into account.

*Table 174 Default values for thin-layer parameters and the numeric integral*

Symbol	Name	Default	Range	Unit
$d_{Ly}$	<code>dLy</code>	3e-1	real	m
$r_\delta$	<code>r_del</code>	1	[0,1]	1
$\eta$	<code>eta</code>	1.5e-10	real	m
$N_\omega$	<code>Nomg</code>	3	integer	1
$\zeta$	<code>order</code>	2	integer	1

## Chapter 33: Thermal Properties

### Thermal Conductivity

## Mole Fraction–Dependent Relaxation Time

To compute  $\kappa$  for alloy materials such as SiGe, the parameters are interpolated according to the standard Vegard law. This is performed using the built-in `MoleFraction`-dependent functionality in Sentaurus Device. All parameters in [Table 170 on page 1022](#) are `MoleFraction`-dependent parameters.

An example of the parameter file is:

```
Kappa {  
    ...  
    omega1(0) = 2.357e13 #[s^-1]  
    omega1(1) = 1.322e13 #[s^-1]  
    ...  
    AI(0) = 1.32e-45 #[s^3]  
    AI(1) = 2.4e-45 #[s^3]  
    Xmax(1) = 1  
    B(AI(1)) = -3.74e-42  
}
```

### Note:

Impurity scattering is significant in alloy materials. Therefore,  $B(AI(1))$  is fitted to achieve good agreement with measurement.

[Equation 1097](#) involves an integral over phonon frequency. The numeric integration is solved with the Gauss quadrature method, which can be fine-tuned by adjusting the following parameters:

- $N_{\omega}$  is the number of integration points.
- $\zeta$  is the order of the Legendre polynomial that is used in the Gauss quadrature method.

[Table 174](#) lists the default values of these two parameters. More integration points and higher orders generally lead to more accurate results, while the performance decreases significantly. Nevertheless, when using  $N_{\omega} = 3$  and  $\zeta = 2$ , the computed  $\kappa$  agrees well with measurements.

---

## The ConnellyThermalConductivity Model

For thin layers, the scattering of acoustic phonons at interfaces reduces the mean free path and reduces thermal conductivities. In the review paper [\[10\]](#), an integral equation describes the geometric effect of the reduced phonon path. The actual model provides an excellent fit of the behavior. The thermal conductivity  $\kappa$  is given by:

$$\kappa = \kappa_{\text{bulk}} \frac{t + t_0}{0.6\Lambda + t + t_0}^{\eta} \quad (1122)$$

## Chapter 33: Thermal Properties

### Thermal Conductivity

where:

- $t$  is the layer thickness extracted by the simulator for the layer.
- $t_0$  is the phonon penetration at interfaces.
- $\Lambda$  is the phonon mean free path.
- $\eta$  is the power exponent for the thickness dependency.
- $\kappa_{\text{bulk}}$  is the bulk thermal conductivity.

Table 175 Parameters of ConnelyThermalConductivity model

Name	Symbol	Default and unit	Range	Description
plot	—	0	{0,1}	Plots parameter to screen
lambda	$\Lambda$	0.3 $\mu\text{m}$	real	Phonon mean free path
t0	$t_0$	5e-4 $\mu\text{m}$	$>=0$	Phonon penetration at interfaces
eta	$\eta$	0.8	real	Power exponent
UseLayerThicknessField	—	0	{0,1}	Selects layer thickness quantity
bulkmodel	—	"Kappa"	string	Name for bulk model parameter

## Layer Thickness Computation

The layer thickness  $t$  is computed internally (see [Extracting Layer Thickness on page 379](#)). By default, the model uses the layer thickness quantity `LayerThickness`. Setting `UseLayerThicknessField=1` selects the quantity `LayerThicknessField`.

## Bulk Thermal Conductivity Computation

The bulk thermal conductivity  $\kappa_{\text{bulk}}$  is computed from parameters found in the section given by the `bulkmodel` parameter. It recognizes the parameters given in [Table 176](#).

Table 176 Parameters of ConnelyThermalConductivity for bulk thermal conductivity

Name	Symbol	Default	Unit	Range	Description
Formula	—	0	—	{0,1}	Selects formula
a1	$a_1$	1.	W/K cm	real	See <a href="#">Equation 1124</a>

## Chapter 33: Thermal Properties

### Thermal Conductivity

Table 176 Parameters of ConnelyThermalConductivity for bulk thermal conductivity

Name	Symbol	Default	Unit	Range	Description
b1	$b_1$	0.	W/K <sup>2</sup> cm	real	
c1	$c_1$	0.	W/K <sup>3</sup> cm	real	
a0	$a_0$	1.	K cm/W	real	See <a href="#">Equation 1123</a>
b0	$b_0$	0.	cm/W	real	
c0	$c_0$	0.	cm/W K	real	

Similar to the default model, for Formula=0:

$$\kappa_{\text{bulk}} = \frac{1}{a_0 + b_0 T + c_0 T^2} \quad (1123)$$

For Formula=1:

$$\kappa_{\text{bulk}} = a_1 + b_1 T + c_1 T^2 \quad (1124)$$

The model is activated in the command file by:

```
Physics (...) {
    ThermalConductivity ( "ConnelyThermalConductivity" )
}
```

### Example of Parameter File Segment

```
Material = "MyMaterial" {
    ConnelyThermalConductivity {
        lambda = 0.3           * [um]
        ...
        UseLayerThicknessField = 0 * use LayerThickness/LayerThicknessField
        bulkmodel = "MyKappa"    * bulk model name
    }
    MyKappa {                  * parameter section specified by 'bulkmodel'
        Formula = 0           * select formula
        ...
    }
}
```

---

## The pmi\_msc\_thermalconductivity Model

This model depends on the lattice temperature and the state occupation probabilities of an MSC. If it does not depend on an MSC, then it allows a pwl temperature dependency and reads as:

$$\kappa = \kappa(T) \quad (1125)$$

If an explicit MSC dependency is given, the global thermal conductivity is computed as:

$$\kappa = \sum_i \kappa_i(T) s_i \quad (1126)$$

where the sum is taken over all MSC states,  $\kappa_i$  are the state thermal conductivities, and  $s_i$  are the state occupation probabilities.

The model is activated in the `Physics` section by:

```
ThermalConductivity (
    PMIModel ( Name="pmi_msc_thermalconductivity" MSConfig="m0" )
)
```

*Table 177 Parameters of pmi\_msc\_thermalconductivity model*

Name	Symbol	Default	Unit	Range	Description
plot	–	0	–	{0,1}	Plot parameter to screen
kappa	$\kappa$	0.	W/Kcm	real	Constant value
kappa_nb_Tpairs	–	0	–	$>= 0$	Number of interpolation points
kappa_Tp<int>_X	–	–	K	real	Temperature at <int>-th interpolation point
kappa_Tp<int>_Y	–	–	W/Kcm	real	Value at <int>-th interpolation point

See [The pmi\\_msc\\_heatcapacity Model on page 1018](#) for an explanation of the individual parameters and how the parameters can be overwritten for individual MSC states. Only the name of the parameters (use `kappa`) and the unit change.

## Thermoelectric Power

Sentaurus Device supports the following choices for computing the thermoelectric powers,  $P_n$  and  $P_p$ , in semiconductors:

- Use a tabulated set of experimental values for silicon as a function of temperature and carrier concentration
- Use analytic formulas with two adjustable parameters,  $\kappa$  and  $s$
- As a user-defined function of the carrier density and the lattice temperature (thermoelectric power PMI)

In metals, the thermoelectric power  $P$  is only allowed as a user-defined function of the electric field vector and the lattice temperature (as a PMI model).

---

## Physical Models

This section discusses the different physical models.

### Table-Based TEPower Model

By default, Sentaurus Device computes the thermoelectric powers,  $P_n$  and  $P_p$ , using a table of experimental values of thermoelectric powers for silicon published by Geballe and Hull [11] as functions of temperature and carrier concentration.

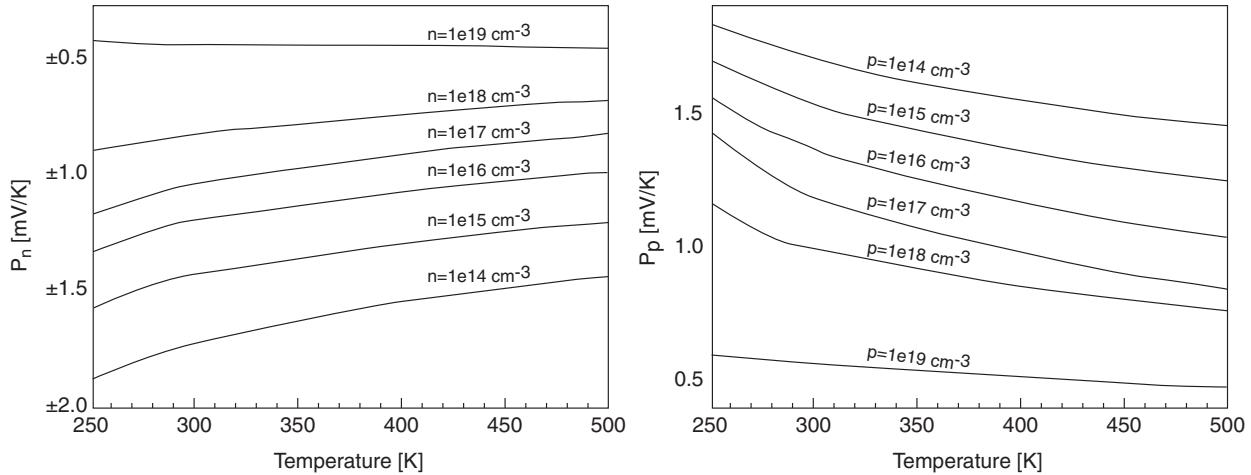
Sentaurus Device extrapolates  $P_n T$  and  $P_p T$  linearly for temperatures between 360 K and 500 K, thereby preserving the  $1/T$  dependency of data presented at higher temperatures by Fulkerson et al. [12], which holds up to near the intrinsic temperature.

$P_n$  and  $P_p$  are shown in [Figure 63](#) as a function of temperature and carrier concentration as used in Sentaurus Device.

## Chapter 33: Thermal Properties

### Thermoelectric Power

**Figure 63** Thermoelectric powers (left)  $P_n$  and (right)  $P_p$  as a function of temperature and carrier concentration



### Analytic TEPower Model

As an alternative, you can use analytic formulas as described in [13][14] to compute the thermoelectric powers in nongenerate semiconductors:

$$P_n = -\kappa_n \frac{k}{q} \left[ \frac{5}{2} - s_n + \ln \frac{N_C}{n} \right] \quad (1127)$$

$$P_p = \kappa_p \frac{k}{q} \left[ \frac{5}{2} - s_p + \ln \frac{N_V}{p} \right] \quad (1128)$$

where you can adjust the parameters  $\kappa$  and  $s$  in the parameter file.

**Table 178** Parameters of analytic TEPower model

Parameter name	Symbol	Default	Unit	Range	Description
s_n	$s_n$	1.	–	real	See <a href="#">Equation 1127</a>
s_p	$s_p$	1.	–	real	See <a href="#">Equation 1128</a>
scale_n	$\kappa_n$	1.	–	real	See <a href="#">Equation 1127</a>
scale_p	$\kappa_p$	1.	–	real	See <a href="#">Equation 1128</a>

## Chapter 33: Thermal Properties

### Thermoelectric Power

## PMI\_ThermoElectricPower Model

In the most general case, thermoelectric powers in semiconductors can be computed using the thermoelectric power PMI (see [Thermoelectric Power on page 1355](#)) as functions of the lattice temperature and the carrier density. Both the standard and simplified PMI (with automatic derivatives) are supported.

## Thermoelectric Power in Metals

In metals, thermoelectric power is defined as a PMI depending on the gradient of the Fermi potential  $\nabla\Phi_M$  and the lattice temperature  $T$  (see [Metal Thermoelectric Power on page 1346](#)). It is used in connection with thermodynamic transport in metals (the Seebeck effect).

---

## Using Thermoelectric Power

Thermoelectric powers in semiconductors are computed automatically when the Temperature equation is solved and the keyword `Thermodynamic` is specified in the global `Physics` section. By default, tabulated silicon data is used.

To activate an analytic formula ([Equation 1127](#), [Equation 1128](#)) or thermoelectric power PMI models either regionwise, or materialwise, or globally, the keyword `TEPower` with `Analytic` or the PMI name as an option must be specified in the corresponding `Physics` section.

The following example activates an analytic formula in "region1", the thermoelectric power PMI `pmi_tepower` in "region2", and tabulated data interpolation in all other semiconductor regions:

```
Physics(Region="region1") {
    TEPower(Analytic)           # analytic formula TEP model in "region1"
}

Physics(Region="region2") {
    TEPower(pmi_tepower)       # PMI in "region2"
}
```

For backward compatibility, the keyword `AnalyticTEP` in the `Physics` section is still available to activate an analytic formula for thermoelectric power globally. It is equivalent to specifying `TEPower(Analytic)` in the global `Physics` section.

The coefficients for the thermoelectric powers defined by the analytic formula are available in the `TEPower` parameter set (see [Table 169 on page 1019](#)).

Thermoelectric power in metals is activated selectively when the Temperature equation is solved and the keyword `Thermodynamic` is specified in the global `Physics` section. To activate it regionwise, materialwise, or globally, the keyword `MetalTEPower` with the metal

## Chapter 33: Thermal Properties

Heating at Contacts, Metal–Semiconductor and Conductive Insulator–Semiconductor Interfaces

thermoelectric power PMI name as an option must be specified in the corresponding Physics sections.

The following example activates the metal thermoelectric power computation and thermodynamic transport in the copper part of the device:

```
Physics(Material="Copper") {
    MetalTEPower(pmi_tepower)
}
```

---

## Heating at Contacts, Metal–Semiconductor and Conductive Insulator–Semiconductor Interfaces

You can model Peltier heat at a contact, or a metal–semiconductor interface, or a conductive insulator–semiconductor interface as:

$$Q = J_n(\alpha_n \Delta E_n + (1 - \alpha_n) \Delta \varepsilon_n) \quad (1129)$$

$$Q = J_p(\alpha_p \Delta E_p + (1 - \alpha_p) \Delta \varepsilon_p) \quad (1130)$$

where:

- $Q$  is the heat density at the contact or interface (when  $Q > 0$ , there is heating; when  $Q < 0$ , there is cooling).
- $J_n$  and  $J_p$  are the electron and hole current densities normal to the contact or interface.
- $\Delta E_n$  and  $\Delta E_p$  are the energy differences for electrons and holes across the interface or at the contact.
- $\alpha_n$ ,  $\alpha_p$ ,  $\Delta \varepsilon_n$ , and  $\Delta \varepsilon_p$  are fitting parameters with  $0 \leq \alpha_n, \alpha_p \leq 1$ .

For the thermodynamic model, Sentaurus Device computes the energy differences for electrons and holes as:

$$\Delta E_n = \Phi_M - \beta_n(\Phi_n + \gamma_n T P_n) + (1 - \beta_n) E_C / q \quad (1131)$$

$$\Delta E_p = \Phi_M - \beta_p(\Phi_p + \gamma_p T P_p) + (1 - \beta_p) E_V / q \quad (1132)$$

where  $\beta_n$ ,  $\beta_p$ ,  $\gamma_n$ , and  $\gamma_p$  are fitting parameters.

For the default lattice temperature model and the hydrodynamic model, Sentaurus Device computes the energy differences for electrons and holes as:

$$\Delta E_n = \Phi_M + E_C / q \quad (1133)$$

$$\Delta E_p = \Phi_M + E_V / q \quad (1134)$$

## Chapter 33: Thermal Properties

### References

To activate Peltier heat, specify the keyword `MSPeltierHeat` in the corresponding interface or electrode `Physics` section. For example:

```
Physics(MaterialInterface="Silicon/Metal") {  
    MSPeltierHeat  
    ...  
}
```

or:

```
Physics(Electrode="cathode") {  
    MSPeltierHeat  
    ...  
}
```

The fitting parameters  $\alpha$ ,  $\beta$ ,  $\gamma$ , and  $\Delta\epsilon$  can be specified in the `MSPeltierHeat` parameter set of the region interface or electrode for which Peltier heat is computed:

```
MaterialInterface = "Silicon/Metal" {  
    MSPeltierHeat {  
        alpha   = 1.0 , 1.0    # [1]  
        beta    = 1.0 , 1.0  
        gamma   = 1.0 , 1.0  
        deltaE = 0.0 , 0.0    # [eV]  
    }  
}
```

Their default values are  $\alpha_n = \alpha_p = \beta_n = \beta_p = \gamma_n = \gamma_p = 1$  and  $\Delta\epsilon_n = \Delta\epsilon_p = 0$  eV.

---

## References

- [1] S. M. Sze, *Physics of Semiconductor Devices*, New York: John Wiley & Sons, 2nd ed., 1981.
- [2] D. J. Dean, *Thermal Design of Electronic Circuit Boards and Packages*, Ayr, Scotland: Electrochemical Publications Limited, 1985.
- [3] C. J. Glassbrenner and G. A. Slack, "Thermal Conductivity of Silicon and Germanium from 3°K to the Melting Point," *Physical Review*, vol. 134, no. 4A, pp. A1058–A1069, 1964.
- [4] S. S. Furkay, "Thermal Characterization of Plastic and Ceramic Surface-Mount Components," *IEEE Transactions on Components, Hybrids, and Manufacturing Technology*, vol. 11, no. 4, pp. 521–527, 1988.
- [5] M. G. Holland, "Analysis of Lattice Thermal Conductivity," *Physical Review*, vol. 132, no. 6, pp. 2461–2471, 1963.
- [6] M. Asheghi *et al.*, "Thermal conduction in doped single-crystal silicon films," *Journal of Applied Physics*, vol. 91, no. 8, pp. 5079–5088, 2002.

## Chapter 33: Thermal Properties

### References

- [7] N. Mikoshiba, "Model for the Metal–Nonmetal Transition in Impure Semiconductors," *Reviews of Modern Physics*, vol. 40, no. 4, pp. 833–838, 1968.
- [8] D. Fortier and K. Suzuki, "Effect of P Donors on Thermal Phonon Scattering in Si," *Le Journal de Physique*, vol. 37, pp. 143–147, February 1976.
- [9] E. H. Sondheimer, "The mean free path of electrons in metals," *Advances in Physics*, vol. 50, no. 6, pp. 499–537, 2001.
- [10] A. M. Marconnet, M. Asheghi, and K. E. Goodson, "From the Casimir Limit to Phononic Crystals: 20 Years of Phonon Transport Studies Using Silicon-on-Insulator Technology," *Journal of Heat Transfer*, vol. 135, no. 6, p. 061601, 2013.
- [11] T. H. Geballe and G. W. Hull, "Seebeck Effect in Silicon," *Physical Review*, vol. 98, no. 4, pp. 940–947, 1955.
- [12] W. Fulkerson *et al.*, "Thermal Conductivity, Electrical Resistivity, and Seebeck Coefficient of Silicon from 100 to 1300°K," *Physical Review*, vol. 167, no. 3, pp. 765–782, 1968.
- [13] R. A. Smith, *Semiconductors*, Cambridge: Cambridge University Press, 2nd ed., 1978.
- [14] C. Herring, "The Role of Low-Frequency Phonons in Thermoelectricity and Thermal Conduction," in *Semiconductors and Phosphors: Proceedings of the International Colloquium*, Garmisch-Partenkirchen, Germany, pp. 184–235, August 1956.

# 34

## Light-Emitting Diodes

---

*This chapter describes the physics and the models used in light-emitting diode simulations.*

### Note:

LED simulations present unique challenges that require problem-specific model and numerics setups. Contact TCAD Support for advice if you are interested in simulating LEDs (see [Contacting Your Local TCAD Support Team Directly](#)).

---

## Modeling Light-Emitting Diodes

From an electronic perspective, light-emitting diodes (LEDs) are similar to lasers operating below the lasing threshold. Consequently, the electronic model contains similar electrothermal parts and quantum-well physics as in the case of a laser simulation.

The key difference between an LED and a laser is a resonant cavity design for lasers that enhances the coherent stimulated emission at a single frequency (for each mode). An LED emits a continuous spectrum of wavelengths based on spontaneous emission of photons in the active region. However, an alternative design for LEDs with a resonant cavity – the resonant cavity LED (RCLED) – uses the resonance characteristics to cause an amplified spontaneous emission in a narrower spectrum to allow for superbright emissions.

The simulation of LEDs presents many challenges. The large dimension of typical LED structures, in the range of a few hundred micrometers, prohibits the use of standard time-domain electromagnetic methods such as finite difference and finite element. These methods require at least 10 points per wavelength and typical emissions are of the order of 1  $\mu\text{m}$ . A quick estimate gives a necessary mesh size in the order of 10 million mesh points for a 2D geometry. Alternatively, the use of the raytracing method approximates the optical intensity inside the device as well as the amount of light that can be extracted from the device. In many cases, a 2D simulation is not sufficient and a 3D simulation is required to give an accurate account of the physical effects associated with the geometric design of the LED.

Innovative designs such as inverted pyramid structures, chamfering of various corners, surface roughening, and drilling holes are performed in an attempt to extract the maximum

## Chapter 34: Light-Emitting Diodes

### Coupling Electronics and Optics in LED Simulations

amount of light from the device. The device editor Sentaurus Structure Editor is well equipped to create complex 3D devices and provides great versatility in exploring different realistic LED designs.

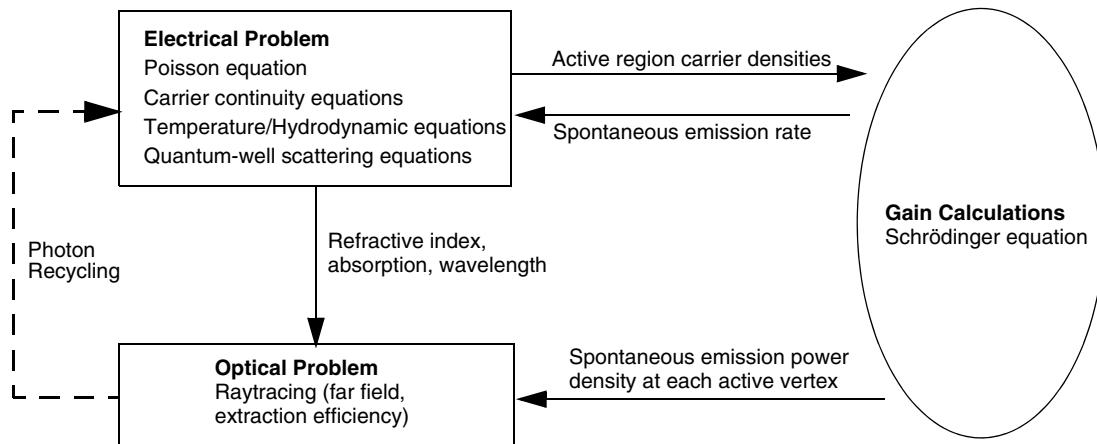
Photon recycling is important because most of the light rays are trapped within the device by total internal reflection. There are two types of photon recycling: for nonactive regions and for the active region. The nonactive-region photon recycling involves absorption of photons in the nonactive regions to produce optically generated electron–hole pairs, and these subsequently join the drift-diffusion processes of the general carrier population. The active-region photon recycling is more complicated, and interested users are referred to [1][2] for its basic theory.

---

## Coupling Electronics and Optics in LED Simulations

An LED simulation solves the Poisson equation, carrier continuity equations, temperature equation, and Schrödinger equation self-consistently. [Figure 64](#) illustrates the coupling of the various equation systems in an LED simulation.

*Figure 64 Flowchart of the coupling between electronics and optics for an LED simulation*



---

## Single-Grid Versus Dual-Grid LED Simulation

Both single-grid and dual-grid LED simulations are possible. However, for single-grid simulations, raytracing takes a longer time because raytracing builds a binary tree for each starting ray. Each branch of the tree corresponds to a ray at a mesh cell boundary. If the materials in two adjoining cells are different, then the ray splits into refracted and reflected rays, creating two new branches. If the materials are the same in adjoining cells, then the propagated ray creates a new branch. A fine mesh increases the depth of the branching significantly. Each new branch of the binary tree is created dynamically and, if dynamic

## Chapter 34: Light-Emitting Diodes

### Electrical Transport in Light-Emitting Diodes

memory allocation of the machine is not sufficiently fast, then the tree creation of the raytracing becomes a bottleneck in the simulation.

To overcome this problem, grids for the electrical problem and the raytracing problem are separated. The optical grid for raytracing is meshed coarsely. The binary tree created will be smaller and raytracing is more efficient. Such a coarse mesh allows you to compute the extraction efficiency and output radiation pattern. However, the optical intensity within the device cannot be resolved well with a coarse mesh.

---

## Electrical Transport in Light-Emitting Diodes

Besides the drift-diffusion transport of typical semiconductor devices, the LED requires additional physical models to compute various optical effects. These physical models are described in the following sections.

---

### Spontaneous Emission Rate and Power

In the active region, the spontaneous emission of photons depletes the carrier population. At each active vertex, the spontaneous emission (or recombination of carriers) rate (units of  $\text{#s}^{-1} \text{m}^{-3}$ ) is an integral of the spontaneous emission:

$$R_{\text{active}}^{\text{sp}}(x, y, z) = \sum_0^{\infty} r^{\text{sp}}(E) \rho^{\text{opt}}(E) dE \quad (1135)$$

where the optical mode density is:

$$\rho^{\text{opt}}(E) = \frac{n_g^2 E^2}{\pi^2 \hbar^3 c^2} \quad (1136)$$

and  $r^{\text{sp}}(E)$  is defined in [Equation 1157 on page 1077](#). The total spontaneous emission power density at each active vertex (units of  $\text{Js}^{-1} \text{m}^{-3}$ ) is:

$$\Delta P^{\text{sp}}(x, y, z) = \sum_0^{\infty} r^{\text{sp}}(E) \rho^{\text{opt}}(E) \cdot (\hbar\omega) dE \quad (1137)$$

This equation is similar to [Equation 1135](#), except that an additional energy term,  $E = \hbar\omega$ , is included in the integrand to account for the energy spectrum of the spontaneous emission.

In LED simulations, the spontaneous emission spectrum,  $r^{\text{sp}}(E)$ , broadens significantly with increasing injected carriers. The integrals in [Equation 1135](#) and [Equation 1137](#) are based on a Riemann sum and, as with numeric integration, truncates at an internally set energy value.

## Chapter 34: Light-Emitting Diodes

### Electrical Transport in Light-Emitting Diodes

You can change the truncation value and the Riemann integration interval with the following syntax in the command file:

```
Physics {...
    LED ...
        Optics ...
            SponScaling = 1.0
            SponIntegration(<energyspan>, <numpoints>)
    }
}
```

where `<energyspan>` is a floating-point number (in eV) and is measured from the edge of the energy bandgap, and `<numpoints>` is an integer denoting the number of discretized intervals to use within this energy span.

The total spontaneous emission power is the volume integral of the power density over all the active vertices:

$$P_{\text{total}}^{\text{sp}} = \int_{\text{(active - region)}} \Delta P^{\text{sp}}(x, y, z) dV \quad (1138)$$

This is the total spontaneous emission power that is computed and output in an LED simulation.

Since you are dealing with a spectrum for the spontaneous emission, it is evident from [Equation 1135](#) and [Equation 1137](#) that the integral sum of the photon rate and the integral sum of the photon power are not simply related by a constant photon energy, that is:

$$\Delta P^{\text{sp}}(x, y, z) \neq (\hbar\omega_0)R_{\text{active}}^{\text{sp}}(x, y, z) \quad (1139)$$

## Spontaneous Emission Power Spectrum

You can plot the LED spontaneous emission power spectrum by activating the `GainPlot` section. The syntax consists of defining the number of discretized points for the spectrum and the span of the spectrum to plot. For example:

```
File {...
    ModeGain = "ngainplot_des"
}
GainPlot {
    Range = (<float>, <float>)      # specific range in eV
    Range = Auto                      # automatically determines range
    Intervals = <int>                # number of discretized points
}
Solve {...}
    PlotGain( Range=(0,1) Intervals=5 )
}
```

In the gain file, the quantity `SponEmissionPowerPereV` (W/eV) is plotted. Integrating this quantity over the energy (eV) span recovers the total LED power.

---

## Current File and Plot Variables for LED Simulations

When an LED simulation is run, specific LED result variables are output to the plot file. [Table 179](#) and [Table 180 on page 1043](#) list the current file output and the plot variables valid for LED simulations.

*Table 179 Current file for LED simulations*

Dataset group	Dataset	Unit	Description
LedWavelength		nm	Average wavelength of LED simulation
n_Contact	Charge	C	
p_Contact	eCurrent	A	
	hCurrent	A	
	InnerVoltage	V	
	OuterVoltage	V	
	TotalCurrent	A	
Photon_Exited		s <sup>-1</sup>	Rate of photon escaping from device
Photon_ExtEfficiency		1	Photon extraction efficiency
Photon_NetPhotonRecycle		s <sup>-1</sup>	Net photon-recycling photon rate
Photon_NonActiveAbsorb		s <sup>-1</sup>	Rate of photon absorption in nonactive regions
Photon_Spontaneous		s <sup>-1</sup>	Spontaneous emission photon rate
Photon_Trapped		s <sup>-1</sup>	Rate of trapped photons in device
Power_Absorption		W	Absorption power
Power_ASE		W	Amplified spontaneous emission power
Power_Exited		W	Power escaping from device
Power_ExtEfficiency		1	Power extraction efficiency
Power_NetPhotonRecycle		W	Net photon-recycling power

## Chapter 34: Light-Emitting Diodes

### Electrical Transport in Light-Emitting Diodes

*Table 179 Current file for LED simulations (Continued)*

Dataset group	Dataset	Unit	Description
Power_NonActiveAbsorb		W	Power absorbed in nonactive regions
Power_ReEmit		W	Reemission power
Power_SpecConvertGain		W	Net power gain of spectral conversion
Power_Spontaneous		W	Spontaneous emission power
Power_Total		W	Total internal optical power of LED
Power_Trapped		W	Power trapped inside device
Time		1	For quasistationary
		s	For transient

*Table 180 Plot variables for LED simulations*

Plot variable	Dataset name	Unit	Description
DielectricConstant		1	Dielectric profile
LED_TraceSource			Influence of each active vertex on the total extracted light
MatGain	OpticalMaterialGain	$\text{m}^{-1}$	Local material gain
RayTraceIntensity		$\text{Wcm}^{-3}$	Optical intensity from raytracing
RayTrees			Raytree structure; not available for the compact memory option
SpontaneousRecombination		$\text{cm}^{-3}\text{s}^{-1}$	Sum of spontaneous emission

`LedWavelength` is computed automatically in the simulation. It is taken as the wavelength where the peak of the spontaneous spectrum occurs. Different options for computing the wavelength are available (see [LED Wavelength on page 1045](#)).

## Chapter 34: Light-Emitting Diodes

### Electrical Transport in Light-Emitting Diodes

For clarity, photon rate and power output results are separated into different groups:

- The photon rate group has units of  $s^{-1}$ .
- The power group has units of W.

Photon and power quantities must be computed separately if a spectrum is involved. Suppose that the spontaneous emission coefficient is  $r^{sp}$  (units of  $eV^{-1} cm^{-3} s^{-1}$ ). The photon rate is  $r^{sp} dE$  (Equation 1135 with  $r^{sp} = r^{sp}(E) \times \rho^{opt}(E)$ ), while the power is  $r^{sp} \cdot E dE$  (Equation 1137).

The extraction coefficient is the ratio of the exited and internal quantities, so the photon rate (`Photon_ExtEfficiency`) and power (`Power_ExtEfficiency`) extraction efficiencies differ.

The only case when `Photon_ExtEfficiency=Power_ExtEfficiency` is for a single wavelength simulation that does not involve a spectrum.

The total outcoupled (exited) power is then (`Power_ExtEfficiency`  $\times$  (`Power_Total`-`Power_NonActiveAbsorb`)), where the total internal optical power is:

```
Power_Total = Power_Spontaneous + Power_NetPhotonRecycle  
+ Power_SpecConvertGain
```

The photon rate extraction efficiency has been computed using:

```
Photon_ExtEfficiency = Photon_Exited / (Photon_Spontaneous  
+ Photon_NetPhotonRecycle  
- Photon_NonActiveAbsorb)
```

For power conservation in a nonactive photon-recycling case, you have:

```
Power_Total = Power_Spontaneous  
= Power_Exited + Power_Trapped + Power_NonActiveAbsorb
```

`Power_Trapped` refers to the power of the photons that are trapped (by total internal reflection or possibly nonconverging raytracing) indefinitely in the raytracing simulation. In a realistic scenario, the trapped photons decay in the device by some mechanism.

#### Note:

Users are responsible for introducing losses within the device (perhaps by introducing nonzero extinction coefficients in appropriate regions) to ensure proper treatment of the trapped photons.

The quantities `Power_NetPhotonRecycle`, `Power_SpecConvertGain`, and `Photon_NetPhotonRecycle` pertain only to the active-region photon-recycling model. These quantities are zero if the active-region photon-recycling model is not activated.

---

## LED Wavelength

There are different ways of computing or inputting the LED wavelength:

- **AutoPeak:** A robust algorithm based on the multisection method is used to search for the peak of the spontaneous emission rate spectrum ( $r^{\text{sp}}$  versus E-curve). Then, the energy of the peak  $r^{\text{sp}}$  is translated into the LED wavelength.
- **AutoPeakPower:** The same algorithm as for AutoPeak is used on the spontaneous emission power spectrum ( $r^{\text{sp}}E$  versus E-curve).
- **Effective:** An effective wavelength is computed such that:  
LED power = LED photon rate × photon\_energy  
where photon\_energy is a direct inverse function of the effective wavelength.
- User inputs a fixed LED wavelength.

The keywords for activating the various LED wavelength options are:

```
Physics {...  
    LED (...  
        Optics (...  
            RayTrace (...  
                Wavelength= <float> | AutoPeak | Effective | AutoPeakPower  
            )  
        )  
    )  
}
```

**Note:**

If no Wavelength keyword is specified, the old peak wavelength search algorithm is used.

---

## Optical Absorption Heat

Photon absorption heat in semiconductor materials can be simplified into two processes:

- **Interband absorption:** When a photon is absorbed across the forbidden band gap in a semiconductor, it is absorbed to create an electron–hole pair. The excess energy (photon energy minus the band gap) of the new electron–hole pair is assumed to thermalize, resulting in eventual lattice heating.
- **Intraband absorption:** A photon can be absorbed to increase the energy of a carrier. The excess energy relaxes eventually, contributing to lattice heating.

In both processes, it is assumed that the eventual lattice heating (in a quasistationary simulation) occurs in the locality of photon absorption.

## Chapter 34: Light-Emitting Diodes

### Electrical Transport in Light-Emitting Diodes

In LEDs, the extraction efficiency ranges between 40% and 60% commonly. This means a significant portion of the spontaneous light power is trapped within the device. The trapped light is ultimately absorbed and becomes the source of photon absorption heat.

Other keywords in the `Physics` section have been added. As far as possible, the syntax has been kept similar to that of the `QuantumYield` model (see [Quantum Yield Models on page 658](#)):

```
Plot {...
    OpticalAbsorptionHeat          # units of [W/cm^3]
}

Physics {...  
    LED(...)  
    OpticalAbsorptionHeat (  
        StepFunction (
            Wavelength = float      # um
            Energy = float         # eV
            Bandgap               # auto-checking of band gap
            EffectiveBandgap       # auto-checking of band gap
        )
        ScalingFactor = float     # default is 1.0
    )
}
```

Some comments about the different `StepFunction` choices:

- When the keyword `Wavelength` or `Energy` is used, the wavelength or energy of the photons is checked against this value. If the photon wavelength or energy is smaller or larger than this cutoff value, then convert the optical generation totally into optical absorption heat, and set optical generation at the vertex to 0.
- `Bandgap` and `EffectiveBandgap` refer to the cutoff energy of the step function at  $E_g - E_{bgn}$  and  $(E_g - E_{bgn} + 2 \cdot (3/2)kT)$ , respectively. If the photon energy is greater than the cutoff energy (interband absorption), then deduct the band gap to obtain the excess energy but do not deduct the optical generation. If the photon energy is less than the cutoff energy (intraband absorption), then set the excess energy to the photon energy. In both cases, the excess energy will be converted to a heat source term for the lattice temperature equation.
- A `ScalingFactor` is introduced to allow for flexible fine-tuning.
- The `OpticalAbsorptionHeat` statement can be specified globally in the `Physics` section or the materialwise or regionwise `Physics` section.

---

## Quantum Well Physics

The physics in the quantum well (QW) is described in detail in [Chapter 35 on page 1076](#). Due to the size of the LED structure, it is recommended that you start with representing

quantum wells as bulk active regions, and then contact TCAD Support for assistance in the more advanced settings (see [Contacting Your Local TCAD Support Team Directly on page 49](#)).

The scattering transport into QWs is approximated with thermionic emission for the best convergence behavior. The corrections of the carrier densities due to quantizations of the QW can be taken into account in the localized QW model (see [Localized Quantum-Well Model on page 1092](#)).

---

## Accelerating Gain Calculations and LED Simulations

The computation bottleneck in LED simulations with gain tables is the calculation of the spontaneous emission rate at every Newton step. The spontaneous emission rate (unit is  $s^{-1}$ ) is an energy integral of the spontaneous emission coefficient (unit is  $s^{-1} eV^{-1} cm^{-3}$ ), and the integration is performed as a Riemann sum. To accelerate this calculation, a Gaussian quadrature integration can be used instead.

All gain calculations (with or without the gain tables) can be accelerated by specifying the following syntax in the `Math` section of the command file:

```
Math { ...
    BroadeningIntegration ( GaussianQuadrature ( Order = 10 ) )
    SponEmissionIntegration ( GaussianQuadrature ( Order = 5 ) )
}
```

The Gaussian quadrature numeric integration then is used in all parts of the gain calculations including broadening effects.

The order can be from 1 to 20. This order refers to the order of the Legendre polynomial that is used to fit the spontaneous emission spectrum in the energy range of the integration. The Gaussian quadrature integration is exact for any spectrum that can be expressed as a polynomial.

**Note:**

You might experience convergence issues in some situations. In such cases, switch off the Gaussian quadrature integration for `SponEmissionIntegration` and `BroadeningIntegration` in the `Math` section.

---

## Discussion of LED Physics

Many physical effects manifest in an LED structure. Current spreading is important to ensure that the current is channeled to supply the spontaneous emission sources at strategic locations that will provide the optimal extraction efficiency.

Changes to the geometric shape of the LED are made to extract more light from the structure. In most cases, the major part of the light produced is trapped within the structure

## Chapter 34: Light-Emitting Diodes

### LED Optics: Raytracing

through total internal reflection. As a result, the nonactive-region and active-region photon-recycling effect becomes relevant. This important physics has been incorporated into different physical models within Sentaurus Device. The nonactive photon-recycling (absorption of photons in nonactive regions) is switched on automatically by default. In most cases, the nonactive photon-recycling model is sufficient to capture the major physical effects because the volume of the active region is insignificant compared to the nonactive regions. Nonetheless, the active-region photon-recycling model can become important if there is a very high stimulated gain, which is usually not the case for GaN LEDs.

Sentaurus Device can easily simulate important aspects of LED design, including current spreading flow, geometric design, and extraction efficiency.

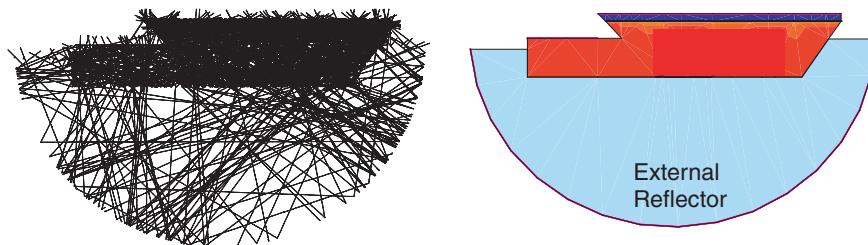
---

## LED Optics: Raytracing

Raytracing is used to compute the intensity of light inside an LED, as well as the rays that escape from the LED cavity to give the signature radiation pattern for the LED output. The basic theory of raytracing is presented in [Raytracing on page 702](#).

Arbitrary boundary conditions can be defined. A detailed description of how to set up the boundary conditions for raytracing is discussed in [Boundary Condition for Raytracing on page 713](#). This is particularly useful in 3D simulations where you can define reflecting planes to use symmetry for reducing the size of the simulation model. In addition, you can use reflecting planes to take into account external components such as reflectors, an example of which is shown in [Figure 65](#).

*Figure 65 Using the reflecting boundary condition to define a reflector for LED raytracing*



The raytracer needs to include the use of the `ComplexRefractiveIndex` model and to define the polarization vector. The syntax for this is:

```
Physics {...
    ComplexRefractiveIndex ...
        WavelengthDep ( real imag )
        CarrierDep( real imag )
        GainDep( real )                      # or real(log)
        TemperatureDep( real )
        CRImodel ( Name = "crimodelname" )
    }
```

## Chapter 34: Light-Emitting Diodes

### LED Optics: Raytracing

```
LED (...  
    Optics (...  
        RayTrace(...  
            PolarizationVector= Random      # or (x y z) vector  
            RetraceCRIchange= float       # fractional change to retrace rays  
            ...  
        )  
    )  
)  
}
```

When you set `PolarizationVector=Random`, random vectors that are perpendicular to the starting ray directions are generated and assigned to be the polarization vector of each starting ray. The direction of each starting ray is described in [Isotropic Starting Rays From Spontaneous Emission Sources](#) and [Anisotropic Starting Rays From Spontaneous Emission Sources on page 1050](#).

The keyword `RetraceCRIchange` specifies the fractional change of the complex refractive index (either the real or imaginary part) from its previous state that will force a total recomputation of raytracing.

---

## Compact Memory Raytracing

A compact memory model is available for LED raytracing. In this model, raytrees are no longer saved, and necessary quantities are computed as required and extracted to compact storages of optical generation and optical intensity. As a result, the memory use and footprint are significantly reduced, thereby allowing the raytracing simulation of large LED structures. The syntax for activation is:

```
Physics {...  
    LED (...  
        RayTrace(...  
            CompactMemoryOption  
        )  
    )  
}
```

### Note:

The full active photon-recycling model does not work with the compact memory model.

---

## Isotropic Starting Rays From Spontaneous Emission Sources

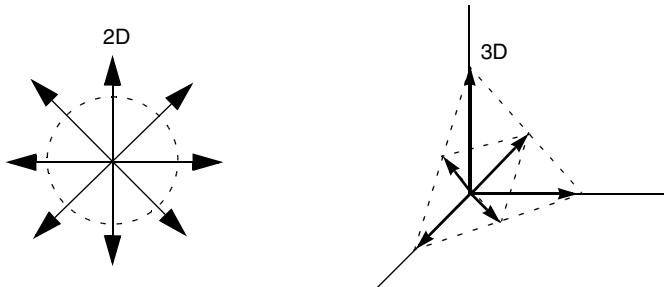
The source of radiation from an LED is mainly from spontaneous emissions in the active region, which is further discussed in [Spontaneous Emission Rate and Power on page 1040](#). The spontaneous emission in the active region of an LED is assumed to be an isotropic

## Chapter 34: Light-Emitting Diodes

### LED Optics: Raytracing

source of radiation and can be conveniently represented by uniform rays emitting from each active vertex, as shown in [Figure 66](#).

**Figure 66** Uniform rays radiating isotropically from an active vertex source in (left) 2D space and (right) 3D space: only one-eighth of spherical space is shown for the 3D case



Isotropy requires that the surface area associated with each ray must be the same. The isotropy of the rays in 2D space is apparent. In 3D space, achieving isotropy is not as simple as dividing the angles uniformly. The elemental surface area of a sphere is  $r^2 \sin\theta(d\theta)(d\phi)$ , so uniformly angular-distributed rays are weighted by  $\sin\theta$  and, therefore, do not signify isotropy.

To approximate this problem in three dimensions, a geodesic dome approximation is used (the geodesic dome is not strictly isotropic). Rays are directed at the vertices of the geodesic dome. The algorithm starts by constructing an octahedron and, then, recursively splits each triangular face of the octahedron into four smaller triangles.

[Figure 66 \(right\)](#) shows the first stage of this splitting process, where rays are directed at the vertices of each triangle. The minimum number of rays is six, that is, one is directed along each positive and negative direction of the axes. If the first stage of recursive splitting is applied, a few more rays are constructed as shown in [Figure 66](#), and the number of starting rays becomes 18. The second stage of recursive splitting gives 68 rays and so on. Therefore, you are constrained to selecting a fixed set of starting rays in the 3D case. Alternatively, you can input your own set of isotropic starting rays (see [Reading Starting Rays From File on page 1052](#)).

---

## Anisotropic Starting Rays From Spontaneous Emission Sources

In some LED designs, the geometry governs the polarization of the optical field in the device. The spontaneous gain is dependent on the direction of this polarization. Consequently, this leads to an anisotropic spontaneous-emission pattern at the source.

The following parametric equations are proposed to describe the anisotropic emission pattern:

$$E_x = d1 \cdot \sin(\phi) + d4 \cdot \cos(\phi) \quad (1140)$$

## Chapter 34: Light-Emitting Diodes

### LED Optics: Raytracing

$$E_y = d2 \cdot \sin(\phi) + d5 \cdot \cos(\phi) \quad (1141)$$

$$E_z = d3 \cdot \sin(\theta) + d6 \cdot \cos(\theta) \quad (1142)$$

where the intensity is given by  $I = E_x^2 + E_y^2 + E_z^2$ .

The bases of sine and cosine are chosen based on the fact that the optical matrix element has such a functional form when polarization is considered (see [Importing Gain and Spontaneous Emission Data With PMI on page 1096](#)). By changing the values of  $d1$  to  $d6$ , different emission shapes can be orientated in different directions, and this feature allows you to modify the anisotropy of the spontaneous emission.

The syntax required to activate the anisotropic spontaneous emission feature is:

```
Physics {...
    LED (...  

        Optics(...  

            RayTrace(...  

                EmissionType(  

                    #Isotropic           # default  

                    Anisotropic(  

                        Sine(d1 d2 d3)  

                        Cosine(d4 d5 d6)
                    )
                )
            )
        )
    )
}
```

---

## Randomizing Starting Rays

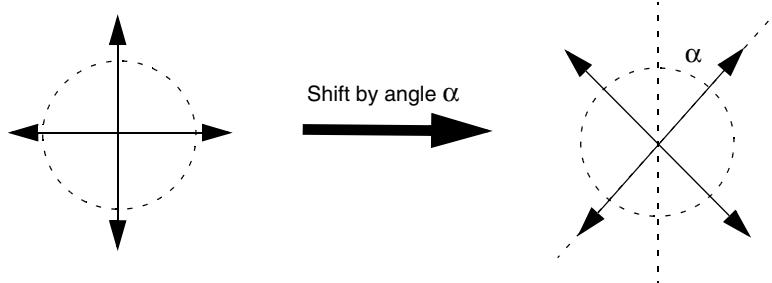
Spontaneous emission is a random process. To take into account the random nature of this process and still ensure that the emission of the starting rays from each active vertex source is isotropic, a randomized shift of the entire isotropic ray emission is introduced.

This is best illustrated in [Figure 67](#) where only four starting rays are used for clarity. For each active vertex, a random angle is generated to determine the random shift of the distribution of the isotropic starting rays. The same concept is also used for the 3D case, and this gives a simple randomization strategy for using raytracing to model the spontaneous emissions.

## Chapter 34: Light-Emitting Diodes

### LED Optics: Raytracing

Figure 67 Shifting the distribution of entire isotropic starting rays by an angle  $\alpha$



## Pseudorandom Starting Rays

Randomization of the starting rays is activated by the keyword `RaysRandomOffset` in the following syntax:

```
Physics {...
    LED (...  

        RayTrace (...  

            RaysRandomOffset  

#           RaysRandomOffset (RandomSeed = 123) # seeding the generator
        )
    )
}
```

You also can fix the seed of the random number generator, and this will compute a pseudorandom set of starting rays, so that repeated runs of the same simulation will reproduce exactly the same raytracing results.

### Note:

Without the keyword `RaysRandomOffset`, the default is a fixed angular shift that is determined by the active vertex number.

---

## Reading Starting Rays From File

The geodesic ray distribution is not truly isotropic. As a result, some percentages of error are incurred when isotropy is really needed. To circumvent this issue, a new feature to allow you to read in a set of source isotropic starting rays has been implemented. The syntax is:

```
Physics {...
    LED (...  

        Optics (...  

            RayTrace (...  

                RaysPerVertex = 1000
                SourceRaysFromFile("sourcerays.txt")
            )
        )
}
```

## Chapter 34: Light-Emitting Diodes

### LED Optics: Raytracing

```
    )  
}
```

It is important to note that `RaysPerVertex` specifies the number of direction vectors to read from the file. The file specified with `SourceRaysFromFile` contains 3D direction vectors for each starting ray; an example of which is `sourcerays.txt`:

```
-0.239117 0.788883 0.566115  
0.776959 0.548552 -0.308911  
-0.607096 -0.042603 0.793485  
-0.158313 -0.276546 -0.947871  
0.347036 -0.805488 0.480370  
...
```

There are several methods for generating 3D isotropic rays, such as the constrained centroid Voronoï tessellation. On the other hand, you can also use this option to import experimentally measured ray distribution profiles.

---

## Moving Starting Rays on Boundaries

Starting rays from active vertices on boundaries create the problem that half of those rays are propagated directly out of the device. To alleviate this problem, a new feature is implemented to allow you to shift the starting position of such rays inwards of the device. Typical values used are from 1 to 5 nm. The syntax is:

```
Physics {...  
    LED (...  
        Optics (...  
            RayTrace (...  
                MoveBoundaryStartRays(float)    # [nm]  
            )  
        )  
    )  
}
```

---

## Clustering Active Vertices

In 3D LED simulations, the number of active vertices can grow significantly when the electrical mesh is refined. This directly implies that the number of starting rays for raytracing increases significantly because that is a direct function of the number of active vertices.

Consequently, the resultant raytree is an exponential function of the number of starting rays, and this results in a very large raytracing problem, which can derail the simulation time and, in part, the memory usage (since the compact memory model declares storage arrays of the size of the number of active vertices).

The solution is to group the active vertices into clusters, with each cluster serving as a distributed source of starting rays for raytracing.

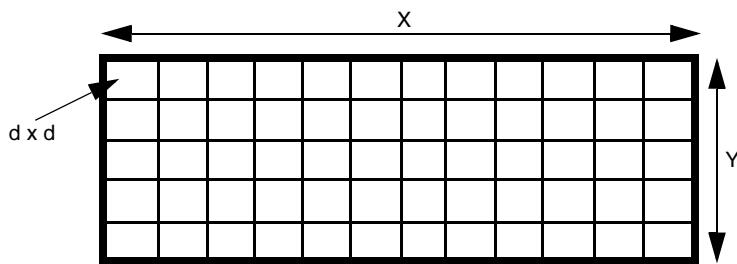
Three possible strategies of clustering the active vertices have been implemented.

## Plane Area Cluster

Users select the total number of clusters to be generated. Then, this number is translated into equal area zones (also taking into account the aspect ratio) of the automatically detected QW plane. The active vertices are subsequently grouped into each of these zones such that each zone forms a cluster. The algorithm for plane area clustering is described as follows.

Assume you input the required cluster size,  $N_c$ . The aim is to fit as many *squarish* elements (of size  $d \times d$ ) into the QW plane area (size  $XY$ ) as possible, as shown in [Figure 68](#).

*Figure 68 Fitting as many squarish elements of size  $d \times d$  into QW plane*



The constraints of the problem are:

$$N_c = \frac{XY}{d^2} \quad (1143)$$

$$X = N_x d, \quad N_x \text{ is an integer} \quad (1144)$$

$$Y = N_y d, \quad N_y \text{ is an integer} \quad (1145)$$

Solving gives:

$$N_x = (\text{INTEGER})\sqrt{(X/Y)N_c} \quad (1146)$$

$$N_y = (\text{INTEGER})(Y/X)N_x \quad (1147)$$

Finally, the adjusted cluster size becomes:

$$N'_c = N_x \times N_y \quad (1148)$$

If no active vertices fall within a plane area segment, that segment is not added to the list of clusters. Therefore, you might see a smaller number of clusters than  $N'_c$ .

## Nodal Clustering

A recursive algorithm is used to calculate how to group the active vertices for each cluster, such that each cluster receives, more or less, the same number of active vertices. The algorithm alternates the x- and y-coordinate partitioning of the list of active vertices, in an attempt to group a cluster of nearest neighboring active vertices. Unfortunately, this might not result in an even spatial distribution of clusters, which is a disadvantage of this method.

## Optical Grid Element Clustering

The number of clusters cannot be set by users as it is defined by the number of optical grid elements. Active regions must be defined in both the electrical and optical grids. An effective bulk region in the optical grid is used to describe the active QW layers and can be meshed according to user requirements. Then, the electrical active vertices are grouped inside each of the optical grid active elements such that each optical-grid active element forms a cluster. In this way, you can control the distribution of the clusters for greater modeling flexibility.

**Note:**

In all these clustering methodologies, the center of each cluster is determined by the average of the active vertices that it encloses.

## Using the Clustering Feature

The following keywords in the LED framework activate the clustering feature:

```
Physics { ...
    LED ( ...
        Raytrace (...
            ClusterActive(
                ClusterQuantity = Nodes | PlaneArea | OpticalGridElement
                NumberOfClusters = <int>           # for Nodes | PlaneArea
            )
        )
    )
}
```

---

## Debugging Raytracing

Rays are assumed, by default, to irradiate isotropically from each active vertex. In the case of quantum wells, an artifact of this assumption might cause unrealistic spikes in the radiation pattern. Consider those source rays that are directed within the plane of the quantum wells. These rays will mostly transmit out of the device at the ending vertical edges of the quantum wells. Realistically, the rays would have a much higher probability of being absorbed and reemitted into another direction than traversing the entire plane of the quantum well.

## Chapter 34: Light-Emitting Diodes

### LED Optics: Raytracing

To circumvent this problem, use the anisotropic emission as described in [Anisotropic Starting Rays From Spontaneous Emission Sources on page 1050](#). Alternatively, one can try to exclude the source rays emitting within a certain angular range from the horizontal plane, that is, the plane of the quantum well. The power from these excluded rays will be distributed equally to the rest of the rays that are not within this angular range. This results in an approximate anisotropic emission shape at each active vertex. The syntax for this exclusion is:

```
Physics { ...
    LED ( ...
        Optics ( ...
            RayTrace ( ...
                ExcludeHorizontalSource(<float>)      # in degrees
            )
        )
    )
}
```

To add more flexibility to LED raytracing, debugging features include:

- Setting a fixed observation center for the LED radiation calculations
- Fixing a constant wavelength for raytracing
- Allowing you to print and track the LED radiation rays (in a certain angular zone) back to its source active vertex

These features are described in this syntax:

```
Physics { ...
    LED ( ...
        Optics ( ...
            RayTrace ( ...
                ObservationCenter = (<float> <float>)
                                # in micrometers, 3 entries for 3D
                Wavelength = 888          # [nm] set fixed wavelength
                DebugLEDRadiation(<filename> <StartAngle> <EndAngle>
                                  <MinIntensity>)
            )
        )
    )
}
```

---

## Print Options in Raytracing

The original `Print` feature of raytracing causes all ray paths to be printed. With multiple reflections or refractions and many starting rays, the resultant image can become a black smudge. To reduce the number of ray paths that are printed, the `skip` option of the `Print` feature is available. In addition, you can trace the ray paths originating from only a single active vertex.

## Chapter 34: Light-Emitting Diodes

### LED Optics: Raytracing

These features are described in this syntax:

```
Physics {
    LED (
        Optics (
            RayTrace (
                Print(Skip(<int>))    # skip printing every <int> ray paths
                Print(ActiveVertex(<int>))    # print only rays from
                                                # active vertex <int>
                PrintSourceVertices(<string>)
                ProgressMarkers = <int>          # 1-100% intervals (integer)
            )
        )
    )
}
```

The option `PrintSourceVertices(<string>)` outputs the list of active vertices, their global index numbering, and coordinates into the file specified by `<string>`. If the number of source rays is large or the optical mesh is fine, then raytracing takes some time to be completed. In this case, use `ProgressMarkers=<int>` to set the incremental completion meter.

The `Print` feature only outputs rays as lines with no other information. To obtain a raytree that contains intensity and other information, you can plot the raytree by using `RayTrees` in the `Plot` section of the command file:

```
Plot {
    RayTrees
}
```

The resultant raytree is plotted in TDR format and can be visualized by Sentaurus Visual, where each branch of the raytree can be accessed individually.

---

## Interfacing LED Starting Rays to LightTools

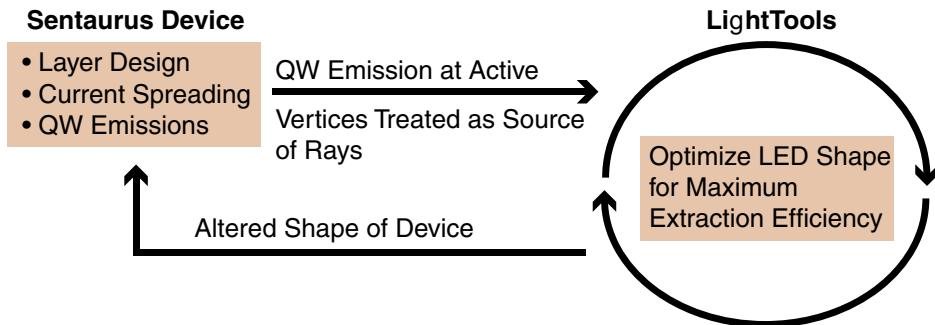
To facilitate the rapid design of LED structures with its luminaire, the starting rays from active region vertices can be output directly to a ray file formatted for use in the Synopsys LightTools® tool. [Figure 69](#) shows a possible design flow.

The set of rays is derived from the LED spontaneous emission power spectrum at each vertex of the device. This means that there is wavelength variability such that each starting ray carries a starting position, a direction, an intensity value, and a wavelength.

## Chapter 34: Light-Emitting Diodes

### LED Optics: Raytracing

Figure 69 LED design flow to optimize extraction efficiency by LightTools and coupled to Sentaurus Device LED device simulation



This feature is an extension of the `Disable` keyword, so that the internal Sentaurus Device raytracing engine will not be activated. The syntax is:

```
File { ...
    Plot = "n99_des.tdr"
}

Physics {...}
    LED (
        RayTrace(
            Disable(
                OutputLightToolsRays (
                    WavelengthDiscretization=<int>
                        # spectrum discretization
                    RaysPerCluster=<int>           # rays emitting from each
                        # active cluster
                    IsotropyType= InBuilt | Random | UserRays
                        # default: InBuilt
                    SaveType= Ascii | Binary # choose ASCII or binary format
                )
            )
            ClusterActive()
            RaysPerVertex=<int>           # used with SourceRaysFromFile()
            SourceRaysFromFile(<string>)
        )
    )
}

Solve {...}
    Plot( Range=(0,1) Intervals=5 )
```

This feature can be used in conjunction with the `ClusterActive` section, so that the active vertices can be grouped into clusters to reduce the final number of rays. If the `ClusterActive` section is not present, every active vertex will be used as emission centers for the starting rays. It is also possible to import an isotropic distribution of point source rays from a file to be used for random-rotated distribution at different vertices. The span of the

## Chapter 34: Light-Emitting Diodes

### LED Optics: Raytracing

spectrum at each active cluster is computed automatically and divided into the numbers as specified by `WavelengthDiscretization`. The total number of starting rays is:

```
WavelengthDiscretization * RaysPerCluster * NumberOfActiveClusters
```

Two types of ray file format for LightTools can be chosen: ASCII or binary. The base name for the ray files is derived from the plot file name and is appended with either `_lighttools.txt` for the ASCII format or `_lighttools.ray` for the binary format.

For example, according to this syntax, the following are the corresponding file names for the LightTools ray files:

```
n99_000000_des_lighttools.txt  
n99_000001_des_lighttools.txt  
...  
n99_000000_des_lighttools.ray  
n99_000001_des_lighttools.ray  
...
```

These files can be read directly by LightTools as volume sources of rays (refer to the LightTools manual for more information).

### Example: n99\_000000\_des\_lighttools.txt File

This example is an ASCII-formatted LightTools ray file (version 2.0 format):

```
# Synopsys Sentaurus Device to LightTools  
# Ray Data Export File  
LT_RDF_VERSION: 2.0  
DATANAME: SentaurusDeviceRayData  
LT_DATATYPE: radiant_power  
LT_RADIANC_FLUX: 1.946806e-04  
LT_FAR_FIELD_DATA: NO  
LT_COLOR_INFO: wavelength  
LT_LENGTH_UNITS: micrometers  
LT_DATA_ORIGIN: 0 0 0  
LT_STARTOFDATA  
0.000000e+00 4.290000e-01 0.000000e+00 -0.163343 -0.986569 0.000000  
5.451892e-05 470.395656  
0.000000e+00 4.290000e-01 0.000000e+00 -0.163343 -0.986569 0.000000  
1.817405e-04 459.664919  
...  
LT_ENDOFDATA
```

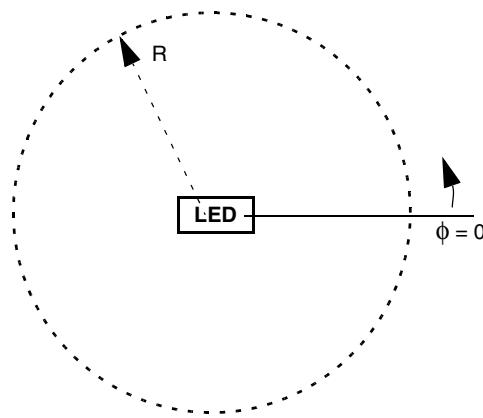
The first three columns denote the starting position (in  $\mu\text{m}$ ) of the ray, the next three columns show the direction vector, which is followed by the fractional power (as a fraction of `LT_RADIANC_FLUX`), and ends with the wavelength (in nm).

## LED Radiation Pattern

Raytracing does not contain phase information, so it is not possible to compute the far-field pattern for LED structures. Instead, the outgoing rays from the LED raytracing are used to produce the radiation pattern.

In 2D space, this is equivalent to moving a detector in a circle around the LED as shown in [Figure 70](#).

*Figure 70 Measuring the radiation pattern in a circular path around LED at observation radius, R*



In 3D space, the detector is moved around on a sphere. The circle and sphere have centers that correspond to the center of the device. Sentaurus Device automatically determines the center of the device to be the midpoint of the device on each axis. Nonetheless, there is an option that allows you to set the center of observation (see [Debugging Raytracing on page 1055](#) and [Table 277 on page 1658](#)).

To examine the optical intensity inside an LED, use `RayTraceIntensity` in the `Plot` section.

The syntax required to activate and plot the LED radiation pattern is located in the `File`, `Physics-LED-Optics-RayTrace`, and `Solve-quasistationary` sections of the command file:

```
File { ...
    # ----- Activate LED radiation pattern and save -----
    LEDRadiation = "rad"
}
...
Physics {... 
    LED (... 
        Optics (... 
            RayTrace(... 
                LEDRadiationPara(1000.0,180) # (radius_micrometers, Npoints)
```

## Chapter 34: Light-Emitting Diodes

### LED Radiation Pattern

```
        ObservationCenter = (2.0 3.5 5.5) # set center
    )
)
}
...
Solve {...  
    # ----- Specify quasistationary -----  
    quasistationary (...  

        PlotLEDRadiation { range=(0,1) intervals=3 }  

        Goal {name="p_Contact" voltage=1.8})  

    {...}  

}
```

The LED radiation plot syntax works in the same way as `GainPlot` (see [Spontaneous Emission Power Spectrum on page 1041](#)) in the `Quasistationary` statement.

An explanation of this example is:

- The base file name, "rad", of the LED radiation pattern files is specified by `LEDRadiation` in the `File` section. The keyword `LEDRadiation` also activates the LED radiation plot.
- Parameters for the LED radiation plot are specified by `LEDRadiationPara` in the `Physics-Optics-RayTrace` section. You must specify the observation radius (in  $\mu\text{m}$ ) and the discretization of the observation circle (2D) or sphere (3D).
- You can set the center of observation by including the keyword `ObservationCenter`. If this is not set, then the center is computed automatically as the middle point of the span of the device on each axis.
- The LED radiation pattern can only be computed and plotted within the `Quasistationary` statement. The keyword `PlotLEDRadiation` controls the number of LED radiation plots to produce.
- The argument `range=(0,1)` in the `PlotLEDRadiation` keyword is mapped to the initial and final bias conditions. In this example, the initial and final (goal) `p_Contact` voltages are 0 V and 1.8 V, respectively. Setting `intervals=3` gives a total of four ( $=3+1$ ) LED radiation plots at 0 V, 0.6 V, 1.2 V, and 1.8 V. In general, specifying `intervals=n` produces  $n + 1$  plots.
- If the LED structure is symmetric, then LED radiation is computed only on a semicircle.

The following sections briefly describe the files that are produced in the LED radiation plot for the 2D and 3D cases.

---

## Two-Dimensional LED Radiation Pattern and Output Files

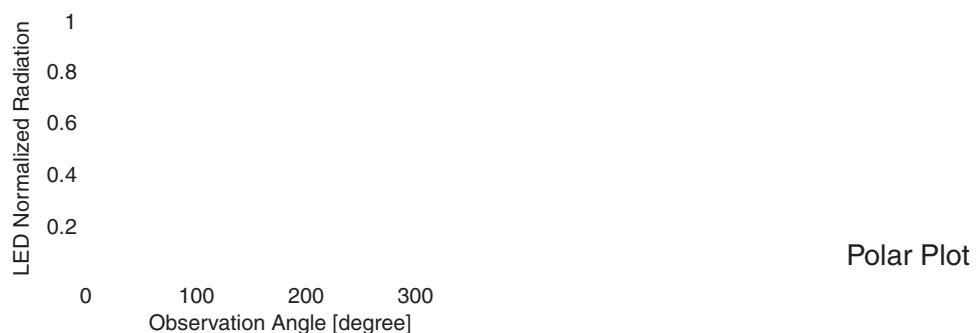
Activating the LED radiation plot for a 2D LED simulation produces two different files (using the base name "rad"):

`rad_000000_LEDRad.plt` The normalized radiation pattern versus observation angle, which can be viewed in Inspect.

`rad_000000_LEDRad_Polar.tdr` The normalized radiation pattern projected onto a grid file and can be viewed in Sentaurus Visual. The polar plot of the LED radiation pattern is then shown.

[Figure 71](#) shows a sample output of the radiation plot of a 2D nonsymmetric LED structure. The lower-left image corresponds to the file `rad_000000_LEDRad.plt` plotted by Inspect, and the right image is the product of the file `rad_000000_LEDRad_Polar.tdr` plotted by Sentaurus Visual.

*Figure 71 (Upper left) LED internal optical intensity, (lower left) normalized radiation intensity versus observation angle, and (right) polar radiation plot computed by Sentaurus Device in 2D LED simulation*



---

## Three-Dimensional LED Radiation Pattern and Output Files

There are two output files for the radiation pattern in the case of a 3D LED simulation:

`rad_000000_des.tdr` Data file containing the normalized radiation pattern. Use Sentaurus Visual for this file, and the spherical plot of the 3D LED radiation pattern is then shown. [Figure 72](#) shows a sample of the radiation pattern of a 3D LED simulation.

## Chapter 34: Light-Emitting Diodes

### LED Radiation Pattern

`rad_000000_des_3Dslices.plt`

The 3D radiation pattern is extracted into polar plots on three planes that can be visualized using Inspect:

- XY plane:  $\theta = \pi/2$ ; plot far field for  $\phi = 0$  to  $2\pi$
- XZ plane:  $\phi = 0, \pi$ ; plot far field for  $\theta = 0$  to  $2\pi$
- YZ plane:  $\phi = \pi/2, 3\pi/2$ ; plot far field for  $\theta = 0$  to  $2\pi$

*Figure 72 (Left) LED internal optical intensity and (right) normalized radiation intensity projected on a sphere of 3D LED simulation*

x                    Radiation on a Sphere            x  
y    y

## Staggered 3D Grid LED Radiation Pattern

In discretizing a 3D far-field collection sphere with regular longitudinal and latitude lines, the size of each surficial element critically depends on its location on the sphere. At polar regions, elements are very small compared to those along the equator. As the discretization increases, the disparity between the polar and equator surficial elements increases at the same rate.

The sampling rate for rays is constrained by the smallest collecting element, which, in this case, is the much smaller surficial element at the poles. In most cases, polar regions are undersampled and equator regions are oversampled due to the disparity in surficial element areas between the two regions. Undersampling at the poles also could lead to anomalous spikes in the far field since the far-field intensity is computed by the total ray power impinging that element and is divided by the elemental area. To adequately sample polar regions, a significant number of rays must be used, resulting in a very large raytracing problem that might not be necessary.

To circumvent these problems, the spherical collection space must be composed of more uniformly distributed elemental areas. The baseline requirement is that each elemental area on the surface of the collection sphere must be approximately the same size.

## Chapter 34: Light-Emitting Diodes

### LED Radiation Pattern

The following simple approach is used:

1. Divide the sphere into concentric rings in the latitude planes. Each ring has a thickness of:

$$d_\theta = R d\theta = R \Delta\theta \quad (1149)$$

2. For each concentric ring, further subdivide the ring into elements with a width that is approximately  $d\theta$ .

Mathematically, assume that the sphere is divided into  $N_\theta$  rings, so that the thickness of each ring is:

$$d_\theta = R \frac{\pi}{N_\theta} \quad (1150)$$

In each concentric ring ( $i$ ) bound between  $\theta_1$  and  $\theta_2$ , choose the order of  $\theta$  such that:

$$\sin(\theta_2) > \sin(\theta_1) \quad (1151)$$

The circumference at  $\theta_2$  is  $C_{\theta_2} = R \sin(\theta_2) \times 2\pi$ . The constraint needed here is such that:

$$\frac{C_{\theta_2}}{N_\phi(i)} = d_\theta \quad (1152)$$

where  $N_\phi(i)$  is the number of divisions for the concentric ring ( $i$ ) and is an integer. Solving gives:

$$N_\phi(i) = (\text{INTEGER})[2 \sin(\theta_2) \times N_\theta] \quad (1153)$$

At the poles of the sphere ( $\theta = 0$  and  $\theta = \pi$ ), the concentric rings collapse into a cap. The following constraint is imposed for these polar cap regions (essentially trying to match triangular area elements with squarish area elements):

$$0.5 \times \frac{C_{\theta_2}}{N_\phi(i)} \times d_\theta = d_\theta^2 \quad (1154)$$

Simplifying gives the number of divisions for the polar cap rings:

$$N_\phi(i) = (\text{INTEGER})[\sin(\theta_2) \times N_\theta] \quad (1155)$$

The total number of elemental areas is  $N_\phi(i)$ .

[Table 181](#) lists the total number of elements obtained from this simple approach.

## Chapter 34: Light-Emitting Diodes

### LED Radiation Pattern

Table 181 Total number of elements and area information as a function of  $N_\theta$

$N_\theta$	Total number of surface elements	Smallest area (unit radius)	Largest area (unit radius)	Area skew = ratio of largest/smallest area
5	36	0.314159	0.399994	1.273
10	146	0.074372	0.097081	1.305
20	554	0.017705	0.024573	1.388
50	3296	0.002857	0.003945	1.381
100	12976	0.000715	0.000987	1.380
150	29013	0.000318	0.000439	1.381
200	51426	0.000179	0.000247	1.380

As  $N_\theta$  increases, it is clear from Table 181 that the area skew (ratio of largest to smallest elemental area) stabilizes to a value of approximately 1.38.

To activate the new staggered 3D far-field grid, use the syntax:

```
Physics {
    LED (
        RayTrace( Staggered3DFarfieldGrid )
    )
}
```

If you do not specify Staggered3DFarfieldGrid, then the collection sphere reverts to the old  $(\theta, \phi)$  scheme to maintain backward compatibility.

---

## Spectrum-Dependent LED Radiation Pattern

Unlike a laser beam with single-frequency emissions, rays emitting from an LED carry a spectrum of frequencies (or energies). Sentaurus Device monitors the spectrum of each ray as it undergoes the process of raytracing in and out of the device. The resultant spectrum of the LED radiation pattern can then be plotted.

To activate this feature, include the keyword `LEDSpectrum` in the command file:

```
Physics ...
    LED ...
        Optics ...
            RayTrace...
                LEDSpectrum(<startenergy> <endenergy> <numpoints>)
```

## Chapter 34: Light-Emitting Diodes

### LED Radiation Pattern

```
)  
)  
}  
}
```

This feature must be used in conjunction with the `LEDRadiation` feature so that the file names of the LED radiation plots and the observation angles can be specified. Other notable aspects of the syntax are:

- `<startenergy>` and `<endenergy>` give the energy range of the spectrum to be monitored. These parameters are floating-point entries with units of eV.
- `<numpoints>` is an integer determining the number of discretized points in the specified energy range.

---

## Tracing Source of Output Rays

Optimizing extraction efficiency is critical in LED designs. Other than modifying the shape of the device, you can use the fact that the rays originating from certain zones in the active region have a higher escape or extraction rate. By designing the shape of the contact to channel higher currents into these zones, the extraction efficiency can be increased effectively.

To facilitate the identification of such zones, a feature is available that correlates the output rays to their source active vertices. The intensity of each output ray is added to an `LED_TraceSource` variable at each active vertex. At the end of the simulation, a profile of `LED_TraceSource` is obtained, which provides an indication of the best localized regions to which more current can be channeled.

The activation of this feature requires keywords to be inserted into two different sections of the command file:

```
Plot { ...  
    LED_TraceSource  
}  
Physics { ...  
    LED ( ...  
        Optics(  
            RayTrace (...  
                TraceSource()  
            )  
        )  
    )  
}
```

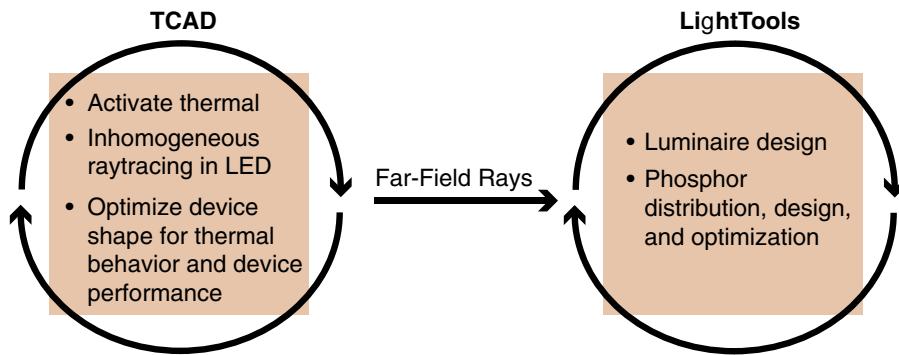
#### Note:

A far-field observation radius must be set for the `TraceSource` feature.

## Interfacing Far-Field Rays to LightTools

LightTools is a robust raytracer that accepts a list of source rays as input, and it can optimize the packaging design for LEDs, that is, the luminaire. [Figure 73](#) shows such a design flow.

**Figure 73** Secondary LED design flow to optimize luminaire design by LightTools and coupled to Sentaurus Device LED device simulation



An interface is available in Sentaurus Device to output farfield rays from an LED device simulation that can be input into LightTools as source rays. This feature requires the LED radiation plot to be activated simultaneously, so that rays can be output at various quasistationary and transient states. The syntax for activating this feature is:

```
Physics {
    LED (
        Optics (
            Raytracer (
                ExternalMaterialCRIFile = "string"
                OutputLightToolsFarfieldRays (
                    Filename = "farfield"
                    WavelengthDiscretization = <int>
                    SaveType = Ascii | Binary
                )
            )
        )
    )
}
```

The energy span of the spectrum is computed automatically, and you only need to input the wavelength discretization. The spectrum information is embedded inside the ray information as wavelength and relative intensity values. The relative intensity is a fraction of the total far-field power.

**Note:**

In Sentaurus Device, raytracing and wavelength dependency of the refractive index can only be specified by using the complex refractive index model.

## Chapter 34: Light-Emitting Diodes

### Nonactive Region Absorption (Photon Recycling)

When the LED is embedded in another medium, the keyword `ExternalMaterialCRIFile` can be included to take wavelength-dependent complex refractive index changes into account (see [External Material in Raytracer on page 723](#)).

You have the option to output the ray information to either an ASCII file or a binary file. The ray information consists of the position vector (x,y,z), the direction cosine (x,y,z), the relative intensity, and the wavelength. Both file types contain identifying headers that are required by LightTools. The resultant name of the file is the user name with the suffix `_lighttools.txt` for the ASCII format or the suffix `_lighttools.ray` for the binary format. For an example of an ASCII-formatted file, see [Example: farfield\\_lighttools.txt File](#).

---

### Example: farfield\_lighttools.txt File

```
# Synopsys Sentaurus Device Farfield to LightTools
# Ray Data Export File
LT_RDF_VERSION: 2.0
DATANAME: SentaurusDeviceFarfieldRayData
LT_DATATYPE: radiant_power
LT_RADIANC_FLUX: 2.066459e-05
LT_FAR_FIELD_DATA: NO
LT_COLOR_INFO: wavelength
LT_LENGTH_UNITS: micrometers
LT_DATA_ORIGIN: 0 0 0
LT_STARTOFDATA
5.351638e+01 2.716777e+02 8.281585e+01 -0.198138 0.651751 0.732094
1.648189e-08 390.559610
3.000000e+02 7.388091e+01 4.290000e-01 0.637854 0.770158 0.000000
4.411254e-04 390.559610
1.001089e+02 3.000000e+02 4.290000e-01 -0.637854 0.770158 0.000000
2.585151e-05 390.559610
0.000000e+00 1.943600e+02 4.290000e-01 -0.637854 -0.770158 0.000000
2.783280e-06 390.559610
1.735878e+02 0.000000e+00 4.290000e-01 0.637854 -0.770158 0.000000
1.631101e-07 390.559610
```

---

### Nonactive Region Absorption (Photon Recycling)

The default setting for LED simulations includes the contribution of absorbed photons in nonactive regions to the continuity equation as a generation rate. This is the basic concept of the nonactive-region photon recycling. In most cases, LEDs are designed with larger bandgap material in nonactive regions to eradicate intraband absorption.

As a result, it might not be necessary to include very small values of nonactive absorption in some situations. To allow for such flexibility, nonactive absorption can be switched off as an option.

The syntax is:

```
Physics { ...
    LED ( ...
        Optics ( ...
            RayTrace ( ...
                NonActiveAbsorptionOff
                OptGenScaling = <float>
                BackgroundOptGen(<float>)      # [#/cm^3]
            )
        )
    )
}
```

If you specify `NonActiveAbsorptionOff`, then the plot variable `OpticalGeneration` still show values, but these values are not included in the continuity equation. A scaling factor, `OptGenScaling`, can be defined to scale the final value of optical generation. You also can set a background optical generation with the keyword `BackgroundOptGen`.

---

## Device Physics and Tuning Parameters

Unlike a laser diode, an LED does not have a threshold current. Therefore, the carriers in the active region are not limited to any threshold value, which means that the spontaneous gain spectrum continues to grow as the bias current increases. The limiting factor for growth is when the QW active region is completely filled and leakage current increases significantly, or dark recombination processes start to dominate with increasing bias and temperature.

The following are main concerns for LED designs:

- Designing the basic layers to optimize the internal quantum efficiency. Polarization charge sheets are needed for AlGaN–GaN–InGaN interfaces, and junction tunneling models might be needed for thin electron-blocking layers. For mature QW technology such as InGaAsP systems, there can be predictive value in advanced  $k \cdot p$  gain calculations for optical gain. However, for difficult-to-grow materials such as InGaN/AlGaN QWs, the uncertainties of growth, mole fraction grading, interface clarity, and so on, make predictive modeling of optical gain almost impossible.
- Extraction efficiency. This is mainly a problem of the geometric shape of the LED structure and the complex refractive index profile of the structure. Temperature, wavelength, and carrier distribution can alter the complex refractive index profile, so the extraction efficiency changes with increasing current injection. Many LED structures have tapered sidewalls to help couple more light out of the device. The slope of the taper can be set as a parameter using Sentaurus Workbench, and the automatic parameter variation feature can be used to optimize the extraction efficiency of the LED geometry.
- Current spreading. You want to spread the current uniformly across the entire active region so that total spontaneous emissions can be increased. In Sentaurus Device, there

is an option to switch off raytracing in an LED simulation. Switching off raytracing only forgoes the extraction efficiency and radiation pattern computation; the total spontaneous emission power is still calculated. This can assist you in the faster optimization of an LED device for uniform current spreading.

- Thermal management, hot spots, how much heat is produced, and how to cool the device. Temperature distribution also affects the mobility and complex refractive index profile and, therefore, impacts the current spreading profile and optical extraction efficiency.

---

## Example of 3D GaN LED Simulation

An LED simulation is best run with a dual-grid approach. The electrical grid must be dense in vicinities where carrier transport details are important. On the other hand, a coarse grid is needed for raytracing. The following is a skeletal sample of command file syntax for a dual-grid 3D GaN LED simulation. Only highlights of the syntax that are important to LED simulations are included.

The typical command file syntax is:

```
#####
## Global declarations ##
#####
File { ... }

Math {
    Digits = 5
    NoAutomaticCircuitContact
    DirectCurrent
    Method = blocked
    # ILS should be chosen for big 3D simulations
    # For 2D, use Pardiso
    Submethod=ILS(set= 5)
    ILSrc=
        set (5) {
            iterative(gmres(100), tolrel=1e-11, tolunprec=1e-4,
                      tolabs=0, maxit=200);
            preconditioning(ilut(1e-9,-1), right);
            ordering(symmetric=nd, nonsymmetric=mpsilst);
            options(compact=yes, linscale=0, fit=5, refinebasis=1,
                    refineresidual=30, verbose=5);
        };
    Derivatives
    Notdamped=20
    Iterations=15
    RelErrControl
    ErrRef(electron)=1e7
    ErrRef(hole)=1e7
    ElementEdgeCurrent
    ExtendedPrecision
```

## Chapter 34: Light-Emitting Diodes

### Device Physics and Tuning Parameters

```
DualGridInterpolation ( Method=Simple )
NumberOfThreads = maximum
Extrapolate
}

#####
## Define the optical solver part ##
#####
OpticalDevice optDevice {
    File {
        Grid = "optic_msh.tdr"
        Parameters = "optparafайл.par"
    }
    RaytraceBC {...}
    Physics {
        ComplexRefractiveIndex (
            WavelengthDep (real imag)
            TemperatureDep(real)
        )
    }
    Physics(Region="EffectiveQW") { Active }
}

#####
## Define the electronic solver part ##
#####
Device elDevice {
    Electrode {...}

    Thermode {...}
        { Name="T_contact" Temperature=300.0 SurfaceResistance=0.05 }

    File {
        Grid = "elec_msh.tdr"
        Parameters = "elecparafайл.par"
        Current = "elsolver"
        Plot = "elsolver"
        LEDRadiation = "farfield"
        Gain = "gainfilename"
    }

    # ----- Choose special LED related plot variables to output -----
    Plot {...}
        RayTraceIntensity
        OpticalGeneration
        LED_TraceSource
        OpticalAbsorptionHeat
        # RayTrees           # not for compact memory option
    }

    # ----- Specify gain plot parameters -----
    GainPlot { ... }
}
```

## Chapter 34: Light-Emitting Diodes

### Device Physics and Tuning Parameters

```
Physics {
    Mobility ()
    EffectiveIntrinsicDensity (NoBandGapNarrowing)
    AreaFactor = 1
    IncompleteIonization
    Thermionic
    Fermi
    RecGenHeat

    OpticalAbsorptionHeat(
        Scaling = 1.0
        StepFunction( EffectiveBandgap )
    )

    # Complex refractive index model needed for raytracer
    ComplexRefractiveIndex (
        WavelengthDep (real imag)
        TemperatureDep (real)
    )

    LED (
        SponScaling = 1      * scale matrix element with this factor
        Optics (
            RayTrace(
                # Disable          # this is for purely electrical investigation
                CompactMemoryOption           # use compact memory model
                Coordinates = Cartesian
                Staggered3DFarfieldGrid

                # ----- Set ray starting and terminating conditions -----
                PolarizationVector = Random
                RaysPerVertex = 20
                RaysRandomOffset (RandomSeed = 123)
                Depthlimit = 100
                MinIntensity = 1e-5

                # ----- Set output options -----
                TraceSource()
                    LEDRadiationPara(10000,60) # (<radius-microns>, Npoints)

                # Choose LED wavelength option
                # Fixed wavelength by entering a <float>, units in [nm]
                Wavelength=AutoPeak      # or Effective or AutoPeakPower
                                         # or <float>
            )
        )
        # ----- Quantum well options -----
        QWTransport
        QEExtension = autodetect
        Strain
        Broadening = 0.04
        Lorentzian
    )
}
```

## Chapter 34: Light-Emitting Diodes

### Device Physics and Tuning Parameters

```
)  
  
# Switch on tunneling for electron blocking layer if necessary  
# eBarrierTunneling "rline1" ()  
  
}  
  
# ----- Define recombination models for non-active regions -----  
Physics (material = "AlGaN") {  
    Recombination (SRH(TempDep) Radiative)  
}  
Physics (material= "GaN") {Recombination (SRH(TempDep) Radiative)}  
  
# ----- Set QWs as active regions ----  
Physics (region="QW1") {  
    Recombination(-Radiative SRH(TempDep)) Active  
}  
...  
Physics (region="QW6") {  
    Recombination(-Radiative SRH(TempDep)) Active  
}  
  
# ----- Include polarization sheet charges between interfaces -----  
Physics (RegionInterface="Window/Cap") {  
    Traps(FixedCharge Conc=-2.666746e+12)  
}  
Physics (RegionInterface="Barrier6/Buffer") {  
    Traps(FixedCharge Conc=-1e+12)  
}  
Physics (RegionInterface="Barrier0/Window") {  
    Traps(FixedCharge Conc=3.8e+12)  
}  
Physics(RegionInterface="Barrier0/QW1") {  
    Traps((FixedCharge Conc=1e12))  
}  
Physics(RegionInterface="QW1/Barrier1") {  
    Traps((FixedCharge Conc=-1e12))  
}  
...  
  
Math {  
# -- Define tunneling nonlocal line for electron blocking layer --  
    NonLocal "rline1" {  
        Barrier(Region = "Window")  
    }  
}  
#####  
## Define mixed-mode system section for dual-grid simulation ##  
#####  
System {  
    elDevice d1 (anode=vdd cathode=gnd) { Physics { OptSolver="opt" } }  
    Vsource_pset drive(vdd gnd) { dc = 2.72 }
```

## Chapter 34: Light-Emitting Diodes

### Device Physics and Tuning Parameters

```
Set ( gnd = 0.0 )
optDevice opt ()
}
#####
## Define solving sequence ##
#####
Solve {
    Coupled (Iterations = 40) { Poisson }
    Coupled (Iterations = 40) { Poisson Electron Hole }
    Coupled { Poisson Electron Hole Contact Circuit }
    Coupled { Poisson Electron Hole Contact Circuit Temperature }
    Quasistationary (
        InitialStep = 0.05
        MaxStep = 0.05
        Minstep = 5e-3
        Plot { range=(0,1) intervals=5 }
        PlotGain { range=(0,1) intervals=5 }
        PlotLEDRadiation { range=(0,1) intervals=5 }
        Goal { Parameter=drive.dc Value=3.8 }
    )
    # Full self-consistent electrical+thermal+optics simulation
    Plugin(breakonfailure) {
        Coupled(Iterations = 20) {Electron Hole Poisson Contact
                                Circuit Temperature}
        Optics
    }
}
}
```

Some comments about the command file syntax:

- Multithreading for the raytracer can be activated by specifying `NumberOfThreads` in the global `Math` section. A value of `maximum` has been chosen in this case so that Sentaurus Device will use the maximum number of threads available on the machine.
- The solver must be ILS for 3D simulations, due to the large matrix that needs to be solved. The parameters for ILS must be tuned for optimal convergence.
- `ExtendedPrecision` is recommended for use in GaN device simulations.
- Various special raytrace boundary conditions can be chosen and are set in the `RayTraceBC` section. For details about these special boundary contacts, see [Boundary Condition for Raytracing on page 713](#).
- Polarization charges at AlGaN–GaN–InGaN interfaces are added using fixed trap charges.
- The wavelength used in raytracing is computed automatically to be the wavelength at the peak of the spontaneous emission spectrum. If a fixed wavelength is required, then you have the option of setting it by using the `Wavelength` keyword.

## Chapter 34: Light-Emitting Diodes

### References

- QW regions must be labeled as `Active` in both the optical and electrical device declarations so that proper internal mapping can be performed. You also must include the keyword `Recombination(-Radiative)` in the QW active regions because the spontaneous emission computation in these regions uses the special LED model and, therefore, the default radiative recombination model should be switched off.
- The `ComplexRefractiveIndex` model must be used in conjunction with the raytracer. In addition, a `PolarizationVector` must be defined with the raytracer.
- When `CompactMemoryOption` is chosen, raytrees are not saved internally, so plotting the `RayTrees` or using `Print(Skip(<int>))` is deactivated.
- Activating the nonlocal tunneling model requires the keyword `eBarrierTunneling` in the `Physics` section and the `NonLocal` line definition in the `Math` section (see [Nonlocal Tunneling at Interfaces, Contacts, and Junctions on page 826](#)).
- The `Plugin` feature is used to create the full self-consistent simulation framework between the electrical, thermal, and optics. To disengage full self-consistency, remark off the `Plugin` and `Optics` statements, in which case, the `Optics` will be solved only once at each bias step.
- The dual-grid electrical and optical feature has been used. If you select a single-grid simulation, then you only need to copy the `Physics-LED` section of this example and insert it into the `Physics` section of a typical single-grid command file.

#### Note:

You can deactivate raytracing in LED simulations by specifying `Disable` in the `RayTrace` statement if you do not require the computation of the extraction efficiency and radiation pattern.

[Table 277 on page 1658](#) lists all the arguments for the LED `RayTrace` option.

---

## References

- [1] W.-C. Ng and M. Pfeiffer, "Generalized Photon Recycling Theory for 2D and 3D LED Simulation in Sentaurus Device," in *6th International Conference on Numerical Simulation of Optoelectronic Devices (NUSOD-06)*, Singapore, pp. 127–128, September 2006.
- [2] W.-C. Ng and G. Létay, "A Generalized 2D and 3D White LED Device Simulator Integrating Photon Recycling and Luminescent Spectral Conversion Effects," in *Proceedings of SPIE, Light-Emitting Diodes: Research, Manufacturing, and Applications XI*, vol. 6486, February 2007.

# 35

## Quantum Wells

---

*This chapter presents the physics of quantum wells, and methods of gain calculations for the quantum wells and bulk regions.*

These gain methods include the simple rectangular well model, and gain physical model interface (PMI). Various gain-broadening mechanisms and strain effects are discussed. A localized quantum well (QW) model is presented together with a simplified InGaN/GaN QW gain model.

---

### Overview

In this section, the focus is on two aspects of modeling the quantum well:

- Radiative recombination processes important in a quantum well
- Gain calculations

A few types of recombination processes are important in the quantum well:

- Auger and Shockley–Read–Hall (SRH) recombinations deplete the QW carriers, and they form the dark current.
- Radiative recombination contains the stimulated and spontaneous recombination processes, which are important processes in LEDs.

These recombinations must be added to the carrier continuity equations to ensure the conservation of particles.

The gain calculation is based on Fermi's golden rule and describes quantitatively the radiative emissions in the form of the stimulated and spontaneous emission coefficients. These coefficients contain the optical matrix element  $|M_{ij}|^2$ , which describes the probability of the radiative recombination processes. In the quantum well, computing the optical matrix element requires knowledge of the QW subbands and QW wavefunctions.

## Chapter 35: Quantum Wells

### Radiative Recombination and Gain Coefficients

Sentaurus Device offers several options for computing the gain spectrum:

- A simple finite well model with analytic solutions. In addition, strain effects and polarization dependence of the optical matrix element are handled separately.
- A localized QW model that uses a trapezoidal QW model to take localized electric field effects into account.
- A simplified InGaN/GaN QW gain model that provides analytic forms to adjust the parameters of the effective masses, various band offsets, and so on. These are subsequently used in the gain calculations.
- A nonlocal QW model that uses the 1D Schrödinger solver on a nonlocal mesh to compute the QW subbands and QW wavefunctions (see [Nonlocal Mesh for the 1D Schrödinger Equation on page 351](#)). This model supports arbitrary potential wells.
- User-specified gain routines in C++ language can be coupled self-consistently with Sentaurus Device using the PMI.

---

## Radiative Recombination and Gain Coefficients

After the carriers are captured in the active region, they experience either dark recombination processes (such as Auger and SRH) or radiative recombination processes (such as stimulated and spontaneous emissions), or escape from the active region. This section describes how stimulated and spontaneous emissions are computed in Sentaurus Device.

---

### Stimulated and Spontaneous Emission Coefficients

In the active region of the LED, radiative recombination is treated locally at each active vertex. The stimulated and spontaneous emissions are computed using Fermi's golden rule.

At each active vertex of the quantum wells, the local stimulated emission coefficient is:

$$r^{\text{st}}(\hbar\omega) = \sum_{i,j} dEC_o k_{\text{st}} |M_{i,j}|^2 D(E) \times (f_i^C(E) + f_j^V(E) - 1) L(E) \quad (1156)$$

and the local spontaneous emission coefficient is:

$$r^{\text{sp}}(\hbar\omega) = \sum_{i,j} dEC_o k_{\text{sp}} |M_{i,j}|^2 D(E) f_i^C(E) f_j^V(E) L(E) \quad (1157)$$

## Chapter 35: Quantum Wells

### Radiative Recombination and Gain Coefficients

where (for general III–V materials):

$$C_0 = \frac{\pi e^2}{n_g c \epsilon_0 m_0^2 \omega} \quad (1158)$$

$$|M_{i,j}|^2 = P_{ij} |O_{i,j}|^2 \frac{m_0}{m_e} - 1 \frac{m_0 E_g (E_g + \Delta)}{12 E_g + \frac{2}{3} \Delta} \quad (1159)$$

$$O_{i,j} = \int_{-\infty}^{\infty} dx \zeta_i(x) \zeta_j^*(x) \quad (1160)$$

$$f_{(i, \Phi_n, E)}^C = \frac{1}{1 + \exp \left[ \frac{E_C + E_i + q\Phi_n + \frac{m_r E}{m_e}}{k_B T} \right]}^{-1} \quad (1161)$$

$$f_{(j, \Phi_p, E)}^V = \frac{1}{1 + \exp \left[ \frac{E_V - E_j + q\Phi_p - \frac{m_r E}{m_h}}{k_B T} \right]}^{-1} \quad (1162)$$

$$D_{(E)}^r = \frac{m_r}{\pi \hbar^2 L_x} \quad (1163)$$

$$m_r = \frac{1}{m_e} + \frac{1}{m_h}^{-1} \quad (1164)$$

$L(E)$  is the gain-broadening function. The electron, light-hole, and heavy-hole subbands are denoted by the indices  $i$  and  $j$ .  $f_{i(E)}^C$  and  $f_{j(E)}^V$  are the local Fermi–Dirac distributions for the conduction and valence bands,  $D_{(E)}^r$  is the reduced density-of-states,  $|O_{i,j}|^2$  is the overlap integral of the quantum mechanical wavefunctions, and  $P_{ij}$  is the polarization-dependent factor of the momentum (optical) matrix element  $|M_{i,j}|^2$ . The spin-orbit split-off energy is  $\Delta$  and  $E_g$  is the bandgap energy. The polarization-dependent factor  $P_{ij}$  in the optical matrix element is set to 1.0 for LED simulations. These emission coefficients determine the rate of production of photons when given the number of available quantum well carriers at the active vertex.

For materials of wurtzite crystal structure (InGaN), the optical matrix element  $|M_{i,j}|^2$  and hole masses are different. These are explained in detail in [Electronic Band Structure for Wurtzite Crystals on page 1082](#).

$k_{st}$  and  $k_{sp}$  are scaling factors for the optical matrix element  $|M_{i,j}|^2$  of the stimulated and spontaneous emissions, respectively. They have been introduced to allow you to tune the stimulated and spontaneous gain curves. Consequently, these parameters can change the threshold current.

## Chapter 35: Quantum Wells

### Radiative Recombination and Gain Coefficients

The activating keywords are `StimScaling` and `SponScaling` in the `Physics-LED` section of the command file:

```
Physics {...
    LED (...  

        Optics (...)  

        # ---- Scale stimulated & spontaneous gain ----  

        StimScaling = 1.0          # default value is 1.0  

        SponScaling = 1.0          # default value is 1.0
    )
}
```

The differential gains for electrons and holes are given by the derivatives of the coefficient of stimulated emission  $r^{\text{st}}(\hbar\omega)$  (see [Equation 1156](#)) with regard to the respective carrier density. They can be plotted by including the keywords `eDifferentialGain` and `hDifferentialGain` in the `Plot` section of the command file.

---

## Active Bulk Material Gain

The stimulated and spontaneous emission coefficients discussed are derived for the quantum well. However, these coefficients can apply to bulk materials with slight modifications. In bulk active materials, it is assumed that the optical matrix element is isotropic. The sum over the subbands is reduced to one electron, and one heavy-hole and one light-hole level, because there is no quantum-mechanical confinement in bulk material. In addition, the subband energies are set to  $E_i = 0$ , and the following coefficients are modified:

$$O_{i,j} = 1 \quad (1165)$$

$$D^r(E) = \frac{1}{2\pi^2} \frac{(2m)^{3/2}}{\hbar^2} E^{1/2} \quad (1166)$$

$$P_{i,j} = 1 \quad (1167)$$

All other expressions remain the same.

---

## Stimulated Recombination Rate

Radiative emissions contribute to the production of photons, but they also deplete the carrier population in the active region. At each active vertex, the stimulated recombination rate of the carriers must be equal to the sum of the photon production rate of every lasing mode so that conservation of particles is ensured. The stimulated recombination rate for each active vertex is:

$$R^{\text{st}}(x, y) = \sum_i r^{\text{st}}(\hbar\omega_i) S_i |\Psi_i(x, y)|^2 \quad (1168)$$

## Chapter 35: Quantum Wells

### Gain-Broadening Models

where the sum is taken over all lasing modes. The stimulated emission coefficient is computed locally at this active vertex and its value is taken at the lasing energy,  $\hbar\omega_i$ , of mode  $i$ .  $S_i$  is the photon rate of mode  $i$ , solved from the corresponding photon rate equation of mode  $i$ , and  $|\Psi_i(x, y)|^2$  is the local optical field intensity of mode  $i$  at this active vertex. This stimulated recombination rate is entered in the continuity equations to account for the correct depletion of carriers by stimulated emissions.

The total stimulated recombination rates on active vertices can be plotted by including the keyword `StimulatedRecombination` in the `Plot` section of the command file.

---

## Spontaneous Recombination Rate

The spontaneous emission rate and power are described in [Spontaneous Emission Rate and Power on page 1040](#). The total spontaneous recombination rates on active vertices can be plotted by including the keyword `SpontaneousRecombination` in the `Plot` section of the command file.

---

## Fitting Stimulated and Spontaneous Emission Spectra

The stimulated and spontaneous emission spectra can be fine-tuned by scaling and shifting. The spectra can be scaled in magnitude by the keywords `StimScaling=<float>` and `SponScaling=<float>` in the `Physics-LED` section. You can also shift the effective emission wavelength (and, therefore, the spectra) by an energy amount specified as `GainShift=<float>`.

---

## Gain-Broadening Models

Different line-shape broadening models are available: Lorentzian, Landsberg, and hyperbolic-cosine. These line-shape functions,  $L(E)$ , are embedded in the radiative emission coefficients in [Equation 1156](#) and [Equation 1157](#) to account for broadening of the gain spectrum.

---

### Lorentzian Broadening

Lorentzian broadening assumes that the probability of finding an electron or a hole in a given state decays exponentially in time [1]. The line-shape function is:

$$L(E) = \frac{\Gamma/(2\pi)}{(E_g - \hbar\omega + E)^2 + (\Gamma/2)^2} \quad (1169)$$

## Landsberg Broadening

The Landsberg model gives a narrower, asymmetric line-shape broadening, and its line-shape function is:

$$L(E) = \frac{(\Gamma(E))/(2\pi)}{(E_g - \hbar\omega + E)^2 + (\Gamma(E)/2)^2} \quad (1170)$$

where:

$$\Gamma(E) = \Gamma \sum_{k=0}^3 a_k \frac{E}{q\Psi_p - q\Psi_n}^k \quad (1171)$$

and  $q\Psi_p - q\Psi_n$  is the quasi-Fermi level separation. The coefficients  $a_k$  are:

$$\begin{aligned} a_0 &= 1 \\ a_1 &= -2.229 \\ a_2 &= 1.458 \\ a_3 &= -0.229 \end{aligned} \quad (1172)$$


---

## Hyperbolic-Cosine Broadening

The hyperbolic-cosine function has a broader tail on the low-energy side compared to Lorentzian broadening, and the line-shape function is:

$$L(E) = \frac{1}{4\Gamma} \cdot \frac{1}{\cosh^2 \frac{|E|}{2\Gamma}} \quad (1173)$$


---

## Syntax to Activate Broadening

You can select *only one* line-shape function for gain broadening. This is activated by the keyword `Broadening` in the `Physics-LED` section of the command file. For example:

```
Physics {...  
    LED (...  
        Optics (...)  
            # --- Line-shape broadening functions, choose one only ----  
            Broadening (Type=Lorentzian Gamma=0.01)  
            # Broadening (Type=Landsberg Gamma=0.01)           # Gamma in [eV]  
            # Broadening (Type=CosHyper Gamma=0.01)  
        )  
    }  
}
```

`Gamma` is the line width  $\Gamma$ , which must be defined in units of eV. If no `Broadening` keyword is detected, Sentaurus Device assumes the gain is unbroadened and does not perform the energy integral in [Equation 1156](#) and [Equation 1157](#).

## Electronic Band Structure for Wurtzite Crystals

To account for the strong coupling of the three valence bands (heavy holes (HH), light holes (LH), and crystal-field split-holes (CH)) in wurtzite crystals, the localized quantum-well model in Sentaurus Device supports a parabolic band approximation using effective masses at the  $\Gamma$ -point. It is derived from the three-band  $\vec{k} \cdot \vec{p}$  method and considers strain effects assuming a growth direction along the  $c$ -axis of the hexagonal lattice. Based on general band-structure parameters for wurtzite crystals and a given strain defined by the mismatch of lattice constants, the band offsets and effective masses parallel and perpendicular to the growth direction are computed.

Starting from the diagonal strain tensor defined as:

$$\varepsilon_{xx} = \varepsilon_{yy} = \frac{a_s - a_0}{a_0} \quad (1174)$$

$$\varepsilon_{zz} = -2 \frac{C_{13}}{C_{33}} \varepsilon_{xx} \quad (1175)$$

$$\varepsilon_{xy} = \varepsilon_{yz} = \varepsilon_{zx} = 0 \quad (1176)$$

where  $a_s$  corresponds to the lattice constant of the substrate, and  $a_0$  is the lattice constant of the unstrained layer. The energies of the valence band edge can be written as [2]:

$$E_{\text{hh}}^0 = E_v^0 + \Delta_1 + \Delta_2 + \theta_\varepsilon + \lambda_\varepsilon \quad (1177)$$

$$E_{\text{lh}}^0 = E_v^0 + \frac{\Delta_1 - \Delta_2 + \theta_\varepsilon}{2} + \lambda_\varepsilon + \sqrt{\frac{\Delta_1 - \Delta_2 + \theta_\varepsilon}{2}^2 + 2\Delta_3^2} \quad (1178)$$

$$E_{\text{ch}}^0 = E_v^0 + \frac{\Delta_1 - \Delta_2 + \theta_\varepsilon}{2} + \lambda_\varepsilon - \sqrt{\frac{\Delta_1 - \Delta_2 + \theta_\varepsilon}{2}^2 + 2\Delta_3^2} \quad (1179)$$

with  $\theta_\varepsilon$  and  $\lambda_\varepsilon$  expressing their dependency on the shear deformation potentials  $D_1$  to  $D_4$ :

$$\theta_\varepsilon = D_3 \varepsilon_{zz} + D_4 (\varepsilon_{xx} + \varepsilon_{yy}) \quad (1180)$$

$$\lambda_\varepsilon = D_1 \varepsilon_{zz} + D_2 (\varepsilon_{xx} + \varepsilon_{yy}) \quad (1181)$$

The strain-dependent conduction band edge is given by:

$$E_c^0 = E_v^0 + \Delta_1 + \Delta_2 + E_g + P_{c\varepsilon} \quad (1182)$$

where the energy shift  $P_{c\varepsilon}$  is due to the hydrostatic deformation potentials parallel ( $a_{cz}$ ) and perpendicular ( $a_{ct}$ ) to the growth direction:

$$P_{c\varepsilon} = a_{cz} \varepsilon_{zz} + a_{ct} (\varepsilon_{xx} + \varepsilon_{yy}) \quad (1183)$$

In the expressions for the band edges (Equation 1177–Equation 1179, and Equation 1182),  $E_v^0$  is used as the reference energy and stands for the CH band-edge energy in the absence of spin-orbit interaction. Table 182 and Table 183 summarize all band structure-related and

## Chapter 35: Quantum Wells

### Gain-Broadening Models

strain-related parameters used in this section along with their specification in the Sentaurus Device parameter file.

*Table 182 Parameters defined in BandstructureParameters section of parameter file*

Symbol	Parameter name	Unit	Description
$A_1$	A1	—	Hole effective mass parameter
$A_2$	A2	—	Hole effective mass parameter
$A_3$	A3	—	Hole effective mass parameter
$A_4$	A4	—	Hole effective mass parameter
$\Delta_{\text{so}}$	so	eV	Spin-orbit split energy
$\Delta_{\text{cr}}$	cr	eV	Crystal-field split energy
$\Delta_1$		eV	Defined as $\Delta_{\text{cr}} = \Delta_1$
$\Delta_2$		eV	Defined as $\Delta_{\text{so}} = 3\Delta_2$
$\Delta_3$		eV	Defined as $\Delta_{\text{so}} = 3\Delta_3$

*Table 183 Parameters defined in QWStrain section of parameter file*

Symbol	Parameter name	Unit	Description
$a_0$	a0	m	Lattice constant at T = 300 K
$\alpha$	alpha	m/K	Model parameters describing linear temperature dependency of lattice constant: $a_0(T) = a_0 + \alpha(T - T_0)$
$T_0$	Tpar	K	
$C_{33}$	C_33	eV	Elastic constant
$C_{13}$	C_13	eV	Elastic constant
$a_{cz}$	a_c	eV	Hydrostatic deformation potential parallel to crystal growth direction (here, $a_{cz}$ and $a_{ct}$ are assumed to be equal)

**Chapter 35: Quantum Wells**  
Gain-Broadening Models

Table 183 Parameters defined in QWStrain section of parameter file (Continued)

Symbol	Parameter name	Unit	Description
$a_{ct}$	a_C	eV	Hydrostatic deformation potential perpendicular to crystal growth direction (here, $a_{cz}$ and $a_{ct}$ are assumed to be equal)
$D_1$	D1	eV	Shear deformation potential
$D_2$	D2	eV	Shear deformation potential
$D_3$	D3	eV	Shear deformation potential
$D_4$	D4	eV	Shear deformation potential

Based on a three-band  $6 \times 6$  Hamiltonian matrix needed to describe the strong coupling of the three valence bands, analytic solutions for the dispersion relations  $E(\vec{k})$  can be derived [2][3][4]. Performing a series expansion of  $E$  at the  $\Gamma$ -point up to the second order in  $k$  yields the effective masses that are used with the localized quantum-well model:

$$m_{hh}^z = -m_0(A_1 + A_3)^{-1} \quad (1184)$$

$$m_{hh}^t = -m_0(A_2 + A_4)^{-1} \quad (1185)$$

$$m_{lh}^z = -m_0 A_1 + \frac{\frac{E_{lh}^0 - \lambda_\epsilon}{E_{lh}^0 - E_{ch}^0}}{A_3} A_3^{-1} \quad (1186)$$

$$m_{lh}^t = -m_0 A_2 + \frac{\frac{E_{lh}^0 - \lambda_\epsilon}{E_{lh}^0 - E_{ch}^0}}{A_4} A_4^{-1} \quad (1187)$$

$$m_{ch}^z = -m_0 A_1 + \frac{\frac{E_{ch}^0 - \lambda_\epsilon}{E_{ch}^0 - E_{lh}^0}}{A_3} A_3^{-1} \quad (1188)$$

$$m_{ch}^t = -m_0 A_2 + \frac{\frac{E_{ch}^0 - \lambda_\epsilon}{E_{ch}^0 - E_{lh}^0}}{A_4} A_4^{-1} \quad (1189)$$

where  $A_1$  to  $A_4$  are called the hole effective mass parameters, which must be specified by users as shown in [Table 182 on page 1083](#).

To activate the simplified model for the treatment of the band structure of wurtzite crystals, the crystal type must be indicated in the `Physics` section of the command file.

## Chapter 35: Quantum Wells

### Gain-Broadening Models

If strain is to be accounted for, then the substrate lattice constant must be specified as a reference for the computation of the strain tensor in each layer as shown here:

```
Physics {
    LED (
        Bandstructure ( CrystalType = Wurtzite )
        Strain ( RefLatticeConst = 3.183e-10 )      #[m]
    )
}
```

Besides the computation of the band-edge energies and the effective masses using the formulas in [Equation 1177–Equation 1179](#), [Equation 1182](#), and [Equation 1184–Equation 1189](#), you can define these quantities explicitly. Each valence band can be characterized by a ladder specification as described in [Explicitly Specifying Ladders on page 353](#). For materials exhibiting the wurtzite crystal structure, the syntax of the explicit ladder specification is extended to classify the specific hole type (HH, LH, or CH). This is needed for labeling the corresponding results such as subband energies or overlapping integrals during visualization.

For example, to explicitly define the properties of the various valence bands considering arbitrary strain, that is, the amount of strain is not declared to the tool, the ladder specification in the `SchroedingerParameters` section reads as follows:

```
SchroedingerParameters {
    Formula = 0,4
    hLadder(0.4376, 1.349, 1, 0.002, HeavyHole)
    hLadder(0.4373, 0.4375, 1, 0.003, LightHole)
    hLadder(0.4376, 0.4378, 1, 0.004, CrystalFieldSplitHole)
}
```

The first and second entry of `hLadder` correspond to the values for the parallel ( $m^z$ ) and the perpendicular ( $m^t$ ) effective mass. The third entry indicates the ladder degeneracy, which is set to 1 for GaN-based semiconductors, and the fourth entry is used to specify the band offset with respect to the valence band edge without strain in eV. To activate the explicit ladder specification, the value of `Formula` for holes must be set to 4.

#### Note:

In this parameter file excerpt of the `SchroedingerParameters` section, the value of `Formula` for electrons is set to 0, which uses the isotropic density-of-states mass as both the quantization mass ( $m^z$ ) and the mass ( $m^t$ ) perpendicular to it.

The keyword `Formula` is used to select one of several options for defining effective masses and band offsets, which are summarized in [Table 184 on page 1086](#).

You can specify a single value, which only affects the specification for holes and uses the default value of 0 for electrons. Using a value pair allows the explicit specification of `Formula` for both electrons (first value) and holes (second value).

*Table 184 Supported values of Formula for localized quantum-well model*

Formula	Description	Limitations
0	Use isotropic density-of-states mass as $m^z$ and $m^t$ .	Only supported for electrons.
2	Use <code>me</code> , <code>mh</code> , and <code>ml</code> to specify relative effective mass for electrons, heavy holes, and light holes, respectively.	Only supported for electrons, light holes, and heavy holes. It cannot be used if <code>NumberOfValenceBands</code> is set to a value greater than 2 in the command file. No distinction between $m^z$ and $m^t$ . Band offsets due to strain cannot be specified explicitly.
4	Use <code>eLadder(...)</code> specification for electrons, and use <code>hLadder(...)</code> specification for holes.	
5	Effective masses and band offsets are computed based on parameters specified in <code>BandstructureParameters</code> and <code>QWStrain</code> sections using parabolic band approximation as described in <a href="#">Electronic Band Structure for Wurtzite Crystals on page 1082</a> .	Only supported for holes.

## Optical Transition Matrix Element for Wurtzite Crystals

To compute the spontaneous emission spectrum,  $r^{sp}(E)$ , in LED simulations as defined in [Equation 1157 on page 1077](#), the optical transition matrix element must be evaluated. For quantum wells grown along the c-axis of the wurtzite crystal, the polarization-dependent transition matrix element for the different conduction band-to-valence band transitions can be written as [5]:

$$|M_{hh}^{TE}|^2 = \frac{3}{2} O_{ij} (M_b^{TE})^2 \quad (1190)$$

$$|M_{lh}^{TE}|^2 = \frac{3}{2} \cos^2(\theta_e) O_{ij} (M_b^{TE})^2 \quad (1191)$$

$$|M_{ch}^{TE}|^2 = 0 \quad (1192)$$

$$|M_{hh}^{TM}|^2 = 0 \quad (1193)$$

$$|M_{lh}^{TM}|^2 = \frac{3}{2} \sin^2(\theta_e) O_{ij} (M_b^{TM})^2 \quad (1194)$$

## Chapter 35: Quantum Wells

### Simple Quantum-Well Subband Model

$$|M_{ch}^{TM}|^2 = \frac{3}{2} O_{ij} (M_b^{TM})^2 \quad (1195)$$

where  $O_{ij}$  refers to the overlap integral between the envelope wavefunctions of the  $i$ -th electron subband and the  $j$ -th valence subband. The anisotropic bulk momentum matrix elements are given by:

$$(M_b^{TE})^2 = \frac{m_0}{6} \frac{1}{m_c^z} - 1 \frac{(E_g + \Delta_1 + \Delta_2)(E_g + 2\Delta_2) - 2\Delta_3^2}{E_g + 2\Delta_2} \quad (1196)$$

$$(M_b^{TM})^2 = \frac{m_0}{6} \frac{1}{m_c^t} - 1 \frac{E_g [(E_g + \Delta_1 + \Delta_2)(E_g + 2\Delta_2) - 2\Delta_3^2]}{(E_g + \Delta_1 + \Delta_2)(E_g + \Delta_2) - \Delta_3^2} \quad (1197)$$

where  $m_c^z$  and  $m_c^t$  denote the relative electron mass parallel and perpendicular to the quantization direction, respectively, and  $m_0$  denotes the electron rest mass. The angle  $\theta_e$  is defined as:

$$k_z = |\vec{k}| \cos(\theta_e) \quad (1198)$$

where  $\vec{k}$  refers to the electron vector, and  $\cos(\theta_e) = 1$  at the G-point of the quantum-well subband.

Arbitrary polarization is modeled as a linear combination of TE and TM polarization according to:

$$|M_{i,j}|^2 = a \cdot |M_{i,j}^{TE}|^2 + (1-a) \cdot |M_{i,j}^{TM}|^2 \quad (1199)$$

where  $a$  is called the `PolarizationFactor`, which can be set in the `QWLocal` section as shown here. For purely TE or TM simulations, it is sufficient to set `Polarization` to the respective identifier:

```
Physics {
    QWLocal (
        Polarization = TE           # TM
        # or
        Polarization = Mixed
        PolarizationFactor = 0.4    # must be in interval [0 1]
    )
}
```

## Simple Quantum-Well Subband Model

This section describes the solution of the Schrödinger equation for a simple finite quantum-well model. This is the default model in Sentaurus Device. This simple quantum-well (QW) subband model is combined with separate QW strain (see [Strain Effects on page 1090](#)) to model most (III–V material) quantum-well systems.

## Chapter 35: Quantum Wells

### Simple Quantum-Well Subband Model

In a quantum well, the carriers are confined in one direction. Of interest are the subband energies and wavefunctions of the bound states, which can be solved from the Schrödinger equation. In this simple QW subband model, it is assumed that the bands for the electron, heavy hole, and light hole are decoupled, and the subbands are solved independently by a 1D Schrödinger equation.

The time-independent 1D Schrödinger equation in the effective mass approximation is:

$$-\frac{\hbar^2}{2} \frac{\partial}{\partial x} \frac{1}{m(x)} \frac{\partial}{\partial x} + V(x) - E^i \zeta^i(x) = 0 \quad (1200)$$

where  $\zeta^i(x)$  is the  $i$ -th quantum mechanical wavefunction,  $E^i$  is the  $i$ -th energy eigenvalue, and  $V(x)$  is the finite well shape potential.

With the following ansatz for the even wavefunctions:

$$\zeta(x) = \begin{cases} C_1 \overset{\circ}{\rightarrow} \cos \left[ \frac{\kappa l}{2} e^{\pm \alpha(|x| - l/2)} \right] & , |x| > l/2 \\ \cos(\kappa x) & , |x| \leq l/2 \end{cases} \quad (1201)$$

and the odd wavefunctions:

$$\zeta(x) = \begin{cases} C_2 \overset{\circ}{\rightarrow} \pm \sin \left[ \frac{\kappa l}{2} e^{\mp (x \mp l/2)} \right] & , |x| > l/2 \\ \sin(\kappa x) & , |x| \leq l/2 \end{cases} \quad (1202)$$

[Equation 1200 becomes \[1\]:](#)

$$\alpha \frac{l}{2} + \frac{m_b}{m_w} \kappa \frac{l}{2} \cot \left[ \frac{\kappa l}{2} \right] = 0 \quad (1203)$$

$$\alpha \frac{l}{2} - \frac{m_b}{m_w} \kappa \frac{l}{2} \tan \left[ \frac{\kappa l}{2} \right] = 0 \quad (1204)$$

with:

$$\kappa = \frac{\sqrt{2m_w E}}{\hbar} \quad (1205)$$

$$\alpha = \frac{\sqrt{2m_b(\Delta E_c - E)}}{\hbar} \quad (1206)$$

The first transcendental equation gives the even eigenvalues, and the second one gives the odd eigenvalues. The wavefunctions are immediately obtained with [Equation 1201](#) and [Equation 1202](#) after the subband energy  $E$  has been computed.

## Chapter 35: Quantum Wells

### Simple Quantum-Well Subband Model

Having obtained the wavefunctions and subband energies, the carrier densities of the 1D-confined system are also computed by:

$$n(x) = N_e^{2D} \sum_i |\zeta_i(x)|^2 F_0(\eta_n - E_i) \quad (1207)$$

$$p(x) = N_{hh}^{2D} \sum_j |\zeta_{j(x)}|^2 F_0(\eta_p - E_{hh}^j) + N_{lh}^{2D} \sum_m |\zeta_{m(x)}|^2 F_0(\eta_p - E_{lh}^m) \quad (1208)$$

where  $F_0(x)$  is the Fermi integral of the order 0, and  $\eta_n$ ,  $\eta_p$  denote the chemical potentials. The indices  $hh$  and  $lh$  denote the heavy and light holes, respectively.

The effective densities of states are:

$$N_e^{2D} = \frac{k_B T m_e}{\hbar^2 \pi L_x} \quad (1209)$$

$$N_{lh/hh}^{2D} = \frac{k_B T m_{lh/hh}}{\hbar^2 \pi L_x} \quad (1210)$$

where  $L_x$  is the thickness of the quantum well. The thickness of each quantum well is automatically detected in Sentaurus Device by scanning the material regions for the keyword `Active`.

The effective masses of the carriers in the quantum well can be changed inside the parameter file:

```
eDOSMass
{
    * For effective mass specification Formula1 (me approximation):
    * or Formula2 (Nc300) can be used :
        Formula = 2      # [1]
    * Formula2:
    * me/m0 = (Nc300/2.540e19)^2/3
    * Nc(T) = Nc300 * (T/300)^3/2
        Nc300      = 8.7200e+16      # [cm-3]
    * Mole fraction dependent model.
    * If just above parameters are specified, then its values will be
    * used for any mole fraction instead of an interpolation below.
    * The linear interpolation is used on interval [0,1].
        Nc300(1)      = 6.4200e+17      # [cm-3]
}
...
SchroedingerParameters:
{ * For the hole masses for Schroedinger equation you can
* use different formulas.
* formula=1 (for materials with Si-like hole band structure)
*   m(k)/m0=1/(A+sqrt(B+C*((xy)^2+(yz)^2+(zx)^2)))
*   where k=(x,y,z) is unit normal vector in reciprocal
*   space. '+' for light hole band, '-' for heavy hole band
* formula=2: Heavy hole mass mh and light hole mass ml are
*   specified explicitly.
```

## Chapter 35: Quantum Wells

### Strain Effects

```
* Formula 2 parameters:  
    Formula = 2      # [1]  
    ml       = 0.027 # [1]  
    mh       = 0.08  # [1]  
* Mole fraction dependent model.  
* If just above parameters are specified, then its values will be  
* used for any mole fraction instead of an interpolation below.  
* The linear interpolation is used on interval [0,1].  
    ml(1)   = 0.094 # [1]  
    mh(1)   = 0.08  # [1]  
}
```

---

## Syntax for Simple Quantum-Well Model

This simple QW subband model is the default model when the `QWTransport` model is activated:

```
Physics {...  
    LED (...  
        Optics (...)  
        # ----- Specify QW model and physics -----  
        QWTransport  
        QWExtension = AutoDetect  # QW widths auto-detection  
    )  
}
```

[Table 271 on page 1655](#) provides the keywords that are associated with this simple QW model.

---

## Strain Effects

It is well known that strain of the quantum well modifies the stimulated and spontaneous emission gain spectra. Due to the deformation potentials in the crystal at the well–bulk interface and valence band mixing effects, band structure modifications occur mainly for the valence bands. They have an impact on the optical recombination and transport properties.

In the simple QW subband model discussed in the previous section, a simpler approach to the QW strain effects is adopted. The simple QW subband model does not include nonparabolicities of the band structure, arising from valence band mixing and strain, in a rigorous manner. However, by carefully selecting the effective masses in the well, a good approximation of the strained band structure can be obtained [\[6\]](#). The effective masses can be changed in the parameter file as previously shown.

Basically, strain has two impacts on the band structure. Due to the deformation potentials, the effective band offsets of the conduction and valence bands are modified.

## Chapter 35: Quantum Wells

### Strain Effects

This is included in the simple QW subband model by:

$$\delta E_C = 2a_c \left(1 - \frac{C_{12}}{C_{11}}\right) \varepsilon \quad (1211)$$

$$\delta E_V^{HH} = 2a_v \left(1 - \frac{C_{12}}{C_{11}}\right) \varepsilon + b \left(1 + 2\frac{C_{12}}{C_{11}}\right) \varepsilon \quad (1212)$$

$$\delta E_V^{LH} = 2a_v \left(1 - \frac{C_{12}}{C_{11}}\right) \varepsilon - b \left(1 + 2\frac{C_{12}}{C_{11}}\right) \varepsilon + \frac{\Delta}{2} - \frac{1}{2}\sqrt{\Delta^2 + 9\delta E_{sh}^2 - 2\delta E_{sh}\Delta} \quad (1213)$$

where  $a_n$  and  $a_c$  are the hydrostatic deformation potential of the conduction and valence bands, respectively. The shear deformation potential is denoted with  $b$  and  $\Delta$  is the spin-orbit split-off energy. The elastic stiffness constants are  $C_{11}$  and  $C_{12}$ , and  $\varepsilon$  is the relative lattice constant difference in the active region. The hydrostatic component of the strain shifts the conduction band offset by  $\delta E_C$  and shifts the valence band offset by  $\delta E_V^0 = 2a_v(1 - C_{12}/C_{11})\varepsilon$ .

The shear component of the strain decouples the light hole and heavy hole bands at the  $\Gamma$  point, and shifts the valence bands by an amount of  $\delta E_{sh} = b(1 + 2C_{12}/C_{11})\varepsilon$  in opposite directions.

---

## Syntax for Quantum-Well Strain

To activate the strain shift, specify the keyword `Strain` in the `Physics-LED` section of the command file:

```
Physics {...
    LED ...
        Optics ....
        # --- QW physics ---
        QWTransport
        QEExtension = AutoDetect
        # --- QW strain ---
        Strain
    }
}
```

The parameters  $a_n$ ,  $a_c$ , and  $b$  can be entered as `a_nu`, `a_c`, and `b_shear`, respectively, in the `QWStrain` section of the parameter file:

```
QWStrain
{
    * Deformation Potentials (a_nu, a_c, b, C_12, C_11
    * and strainConstant eps :
    * Formula:
    * eps = (a_bulk - a_active)/a_active
    * dE_c = ...
    * dE_lh = ...
    * dE_hh = ...
```

## Chapter 35: Quantum Wells

### Localized Quantum-Well Model

```
    eps = -1.0000e-02      # [1]
    * a_nu = 1.27          # [1]
    * a_c = -5.0400e+00   # [1]
    * b_shear = -1.7000e+00 # [1]
    * C_11 = 10.11         # [1]
    * C_12 = 5.61          # [1]
}
```

The elastic stiffness constants  $C_{11}$  and  $C_{12}$  can be specified by  $c_{11}$  and  $c_{12}$ . Due to valence band mixing and strain, the valence bands can become nonparabolic. However, within a small range from the band edge, parabolicity can still be assumed. In the simulation, you can modify the effective heavy-hole and light-hole masses in the parameter file for the subband calculation to account for this effect.

The spin-orbit split-off energy can be specified in the `BandstructureParameters` section of the parameter file:

```
BandstructureParameters{
    ...
    so = 0.34    # [eV]
    ...
}
```

---

## Localized Quantum-Well Model

### Note:

This is an advanced model. If you are interested in using this model, then contact TCAD Support for assistance in evaluating whether this model is suitable for use in your device (see [Contacting Your Local TCAD Support Team Directly](#)).

In GaN-based quantum-well systems, the polarization charge sheets at the interfaces of the quantum well or barrier induce a large field within the quantum well. These fields skew the energy bands and cause mismatches in the alignment of the electron and hole wavefunctions.

The localized quantum-well model takes into account field effects in a fully coupled methodology. The activation syntax is included in the `Physics` section of each active quantum well:

```
Physics (region="QW1") {...  
    Active(Type=QuantumWell)  
    QWLocal (  
        NumberOfElectronSubbands = 5  
        NumberOfLightHoleSubbands = 2  
        NumberOfHeavyHoleSubbands = 4  
        NumberOfCrystalFieldSplitHoleSubbands = 3  
        NumberOfValenceBands = 3  
        # -ElectricFieldDep
```

## Chapter 35: Quantum Wells

### Localized Quantum-Well Model

```
    WidthExtraction (
        # indicate side regions or materials of QW if sides
        # do not coincide with domain boundaries
        SideRegion = ("reg1", ..., "regn")
        SideMaterial = ("mat1", ..., "matn")
        MinAngle = (<float>, <float>)
        ChordWeight = <float>
    )
}
}
```

By default, the electric field dependency is activated in the localized quantum-well model. However, it can be deactivated by using the `-ElectricFieldDep` keyword. The maximum number of subbands for electrons, heavy holes (HH), light holes (LH), and crystal-field split-holes (CH) must be specified to enable Sentaurus Device to limit the scope of the computation. The actual number of subbands used is computed as the simulation progresses. By default, only the heavy-hole (HH) band and the light-hole (LH) band are considered. Changing the value of `NumberOfValenceBands` from 2 to 3 includes the crystal-field split-hole (CH) band in the computation and is the recommended setting for materials exhibiting the wurtzite crystal system.

A `WidthExtraction` section is included to enable you to specify how the quantum-well thickness can be extracted. This thickness is important to define the quantum-well width so as to compute the solution to the Schrödinger equation. `MinAngle` and `ChordWeight` are parameters used in a special method to compute the thickness of the quantum-well layer that is not aligned to one of the major axes (see [Thickness Extraction on page 382](#)).

If the `QWLocal` section is defined in the global `Physics` section, the parameters of the maximum number of bands in each band are applied to all the specified `Active(Type=QuantumWell)` regions.

By default, the localized quantum-well model does not account for the correction of the densities in the quantum well due to quantization. To take quantization into account, use the `eDensityCorrection` and `hDensityCorrection` options of `QWLocal`. For more details, see [Quantum-Well Quantization Model on page 378](#).

**Note:**

The variables in [Table 185](#) also apply to the nonlocal quantum-well model.

*Table 185 Variables for plotting when using the localized quantum-well model*

Variable	Description
<code>QW_chEigenEnergy</code>	Eigenenergies of the crystal-field split-hole bound states [eV]
<code>QW_chNumberOfBoundStates</code>	Actual number of QW bound states for crystal-field split-holes
<code>QW_chRelativeEffectiveMass</code>	Relative effective mass of crystal-field split-holes

## Chapter 35: Quantum Wells

### Localized Quantum-Well Model

*Table 185 Variables for plotting when using the localized quantum-well model (Continued)*

Variable	Description
QW_chStrainBandShift	Shift in crystal-field split-hole band due to strain effects
QW_eEigenEnergy	Eigenenergies of the electron bound states [eV]
QW_ElectricFieldProjection	Electric field in the QW [V/cm]
<b>Note:</b>	
	This variable does not apply to the nonlocal quantum-well model.
QW_eNumberOfBoundStates	Actual number of QW bound states for electrons
QW_eRelativeEffectiveMass	Relative effective mass of electrons
QW_eStrainBandShift	Shift in conduction band due to strain effects
QW_hhEigenEnergy	Eigenenergies of the heavy-hole bound states [eV]
QW_hhNumberOfBoundStates	Actual number of QW bound states for heavy holes
QW_hhRelativeEffectiveMass	Relative effective mass of heavy holes
QW_hhStrainBandShift	Shift in heavy-hole band due to strain effects
QW_lhEigenEnergy	Eigenenergies of the light-hole bound states [eV]
QW_lhNumberOfBoundStates	Actual number of QW bound states for light holes
QW_lhRelativeEffectiveMass	Relative effective mass of light holes
QW_lhStrainBandShift	Shift in light-hole band due to strain effects
QW_OverlapIntegral	Overlap integrals between electron and hole wavefunctions
QW_QuantizationDirection	Quantization direction of the QW
QW_Width	Extracted width of the QW [ $\mu\text{m}$ ]

You also can include any of the variables in [Table 185](#) in the CurrentPlot statement. For example:

```
CurrentPlot { ...
    QW_eEigenEnergy (
        Minimum (Region = "QW1")
```

## Chapter 35: Quantum Wells

### Nonlocal Quantum-Well Model Using 1D Schrödinger Solver

```
    Maximum (Region = "QW1")
    Average (Region = "QW1")
)
}
```

---

## Nonlocal Quantum-Well Model Using 1D Schrödinger Solver

### Note:

This is an advanced model. If you are interested in using this model, then contact TCAD Support for assistance in evaluating whether this model is suitable for use in your device (see [Contacting Your Local TCAD Support Team Directly](#)).

The most rigorous model available in Sentaurus Device to model quantum wells is the 1D Schrödinger solver on a nonlocal mesh that covers the quantum-well structure (see [Nonlocal Mesh for the 1D Schrödinger Equation on page 351](#)). A nonlocal mesh consists of numerous cutlines parallel to the quantization direction on which the potential distribution and effective masses are extracted. On each nonlocal line, the 1D Schrödinger solver computes the eigenenergies and the wavefunctions. Then, the eigenenergies of the bound states and the overlap integrals of the wavefunctions for the various optical transitions are interpolated back to the device simulation mesh to calculate the radiative recombination.

This model also supports the simulation of mole fraction-graded quantum wells by defining the band-structure parameters and the effective masses as mole fraction dependent (see [Mole-Fraction Specification on page 65](#) and [Parameters for Composition-Dependent Materials on page 72](#)).

To use the 1D Schrödinger solver for quantum wells:

1. Construct a special-purpose *nonlocal* mesh that covers the quantum-well structure (see [Nonlocal Mesh for the 1D Schrödinger Equation on page 351](#)). The nonlocal lines should extend beyond the well region into the barrier regions on either side by approximately the well width.
2. Activate the 1D Schrödinger solver on the nonlocal line mesh with appropriate parameters (see [Using the 1D Schrödinger Equation on page 352](#), [Parameters of the 1D Schrödinger Equation on page 352](#), and [Electronic Band Structure for Wurtzite Crystals on page 1082](#)).
3. Mark quantum-well regions as active by specifying `Active(Type=QuantumWell)` in the corresponding region-specific `Physics` sections.

## Chapter 35: Quantum Wells

### Importing Gain and Spontaneous Emission Data With PMI

For example, to use the 1D Schrödinger solver to simulate a quantum well that is 3 nm wide and whose plane is perpendicular to the z-axis, specify the following syntax in the command file:

```
Physics (Region="QW1") { Active(Type=QuantumWell) }

NonLocal "QW1" (
    RegionInterface="QW1/Barrier0"
    Length = 6e-7
    Permeation = 3e-7
    Direction = (0 0 1)
    MaxAngle = 5
    Discretization = 1e-8
    -Transparent(Region="Barrier0")
)
Physics {
    Schroedinger "QW1" (Electron Hole Polarization=TE -DensityCorrection)
}
```

By default, the nonlocal quantum-well model accounts for the correction of the densities in the quantum well due to quantization. However, at the beginning, it is recommended to disable the correction of the densities (`-DensityCorrection`) as it has an adverse effect on convergence as shown in this example. For details, see [1D Schrödinger Equation on page 351](#).

The variables in [Table 185 on page 1093](#) can be plotted when using the nonlocal quantum-well model. These variables also can be specified in the `CurrentPlot` statement (see [Tracking Additional Data in the Current File on page 165](#)) and the `NonlocalPlot` statement (see [Visualizing the Solutions on page 355](#)).

---

## Importing Gain and Spontaneous Emission Data With PMI

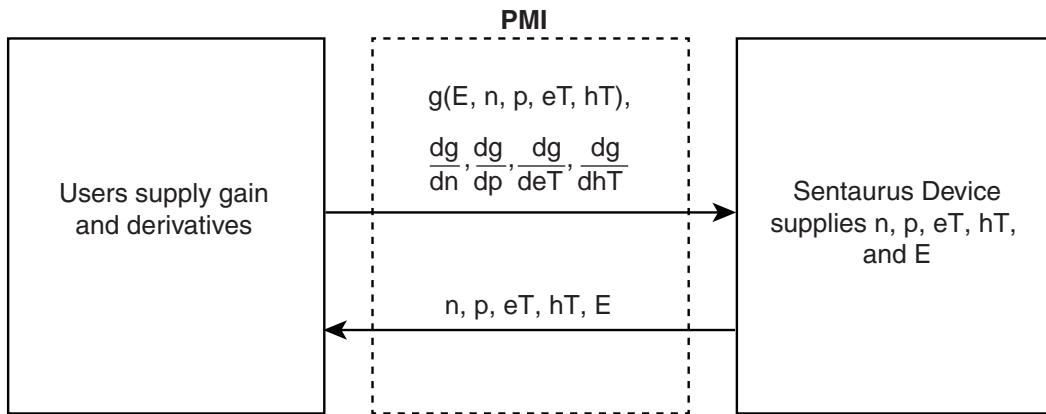
Sentaurus Device can import external stimulated and spontaneous emission data through the physical model interface (PMI).

Sentaurus Device calls the user-written gain calculations through the PMI with the variables: electron density  $n$ , hole density  $p$ , electron temperature  $eT$ , hole temperature  $hT$ , and transition energy  $E$ . The user-written gain calculation then returns the gain  $g$  and the derivatives of gain with respect to  $n$ ,  $p$ ,  $eT$ , and  $hT$  to Sentaurus Device. The derivatives are required to ensure proper convergence of the Newton iterations. In this way, the user-import gain is made self-consistent within the simulation.

## Chapter 35: Quantum Wells

### Importing Gain and Spontaneous Emission Data With PMI

Figure 74 Illustration of the gain PMI concept



---

## Implementing Gain PMI

The PMI uses the object-orientation capability of the C++ language (see [Chapter 39 on page 1207](#)). A brief outline is given here of the gain PMI.

In the Sentaurus Device header file `PMIModels.h`, the following base class is defined for gain:

```
class PMI_StimEmissionCoeff : public PMI_Vertex_Interface {  
public:  
    PMI_StimEmissionCoeff (const PMI_Environment& env);  
    virtual ~PMI_StimEmissionCoeff ();  
  
    virtual void Compute_rstim  
(double E,  
     double n,  
     double p,  
     double et,  
     double ht,  
     double& rstim) = 0;  
  
    virtual void Compute_drstimdn  
(double E,  
     double n,  
     double p,  
     double et,  
     double ht,  
     double& drstimdn) = 0;  
  
    virtual void Compute_drstimdp  
(double E,  
     double n,  
     double p,  
     double et,
```

## Chapter 35: Quantum Wells

### Importing Gain and Spontaneous Emission Data With PMI

```
    double ht,
    double& drstimdpt) = 0;

    virtual void Compute_drstimdpt
    (double E,
     double n,
     double p,
     double et,
     double ht,
     double& drstimdpt) = 0;

    virtual void Compute_drstimdht
    (double E,
     double n,
     double p,
     double et,
     double ht,
     double& drstimdht) = 0;
};

}
```

To implement the PMI model for gain, you must declare a derived class in the user-written header file:

```
#include "PMIModels.h"

class StimEmissionCoeff : public PMI_StimEmissionCoeff {
// User-defined variables for his/her own routines
private:
    double a, b, c, d;

public:
    // Need a constructor and destructor for this class
    StimEmissionCoeff (const PMI_Environment& env);
    ~StimEmissionCoeff ();

    // --- User must write the following routines in the .C file ---
    // The value of the function is return as the last pointer argument

    // stimulated emission coeff value
    void Compute_rstim (double E,
                        double n,
                        double p,
                        double et,
                        double ht,
                        double& rstim);

    // derivative wrt n
    void Compute_drstimdtn (double E,
                            double n,
                            double p,
                            double et,
                            double ht,
                            double& drstimdtn);
```

## Chapter 35: Quantum Wells

### References

```
// derivative wrt p
void Compute_drstimdp (double E,
                        double n,
                        double p,
                        double et,
                        double ht,
                        double& drstimdp);

// derivative wrt eT
void Compute_drstimdet (double E,
                        double n,
                        double p,
                        double et,
                        double ht,
                        double& drstimdet);

// derivative wrt hT
void Compute_drstimdht (double E,
                        double n,
                        double p,
                        double et,
                        double ht,
                        double& drstimdht);

};

}
```

Next, you must write the functions `Compute_rstim`, `Compute_drstimdn`, `Compute_drstimdp`, `Compute_drstimdet`, and `Compute_drstimdht` to return the values of the stimulated emission coefficient and its derivatives to Sentaurus Device using this gain PMI. If you have, for example, a table of gain values, then you must implement these functions to interpolate the values of the gain and derivatives from the table.

The spontaneous emission coefficient can also be imported using the PMI. The implementation is exactly the same as the stimulated emission coefficient, and you only need to replace `StimEmissionCoeff` with `SponEmissionCoeff` in the previous code example.

---

## References

- [1] L. A. Coldren and S. W. Corzine, *Diode Lasers and Photonic Integrated Circuits*, New York: John Wiley & Sons, 1995.
- [2] S. L. Chuang and C. S. Chang, "A band-structure model of strained quantum-well wurtzite semiconductors," *Semiconductor Science and Technology*, vol. 12, no. 3, pp. 252–263, 1997.
- [3] S. L. Chuang and C. S. Chang, "k·p method for strained wurtzite semiconductors," *Physical Review B*, vol. 54, no. 4, pp. 2491–2504, 1996.

## Chapter 35: Quantum Wells

### References

- [4] M. Kumagai, S. L. Chuang, and H. Ando, "Analytical solutions of the block-diagonalized Hamiltonian for strained wurtzite semiconductors," *Physical Review B*, vol. 57, no. 24, pp. 15303–15314, 1998.
- [5] S. L. Chuang, "Optical Gain of Strained Wurtzite GaN Quantum-Well Lasers," *IEEE Journal of Quantum Electronics*, vol. 32, no. 10, pp. 1791–1800, 1996.
- [6] Z.-M. Li *et al.*, "Incorporation of Strain Into a Two-Dimensional Model of Quantum-Well Semiconductor Lasers," *IEEE Journal of Quantum Electronics*, vol. 29, no. 2, pp. 346–354, 1993.

# 36

## Kinetic Monte Carlo MIM Transport

---

*This chapter describes how to study transport in metal–insulator–metal structures at the microscopic level by using an interface to a kinetic Monte Carlo simulator that accounts for transport mechanisms in thin insulating films.*

---

### KMC Simulation Space and Math Settings

In the Sentaurus Device command file, the `Math` section settings that relate specifically to the kinetic Monte Carlo metal–insulator–metal (KMC-MIM) capability are:

```
Math {
    KMC (
        MinLocation = (<xmin>, <ymin>, <zmin>) # [um]
        MaxLocation = (<xmax>, <ymax>, <zmax>) # [um]
        CellLength = <float>                      # [um] (for TDDB only)
        [HalfCellOffset]                            # [1], default true (for TDDB only)
        [ReferenceElectrode = "<string>"]          # [1]
        [Trap2ElecMinDist = <float>]                # [um]
        [Trap2TrapMinDist = <float>]                # [um]
        [RandomSeed = <int>]                       # [1]
        [SkipKMCCurrentCheck = <float>]            # [1]
        [SkipKMCNumber = <int>]                     # [1]
        [SkipKMC]                                # [1]

        [ResetElectronFill]                         # [1]
        [UseAllCrossings]                          # [1]
        [MoreFrequentRateResets]                  # [1]

        [MIMEEventCountCheckPoint = <int>]         # [1]
        [MIMBackAndForthThreshold = <int>]         # [1]
        [IntegrationPoints = <int>]                # [1]

        [ApproximateBessel]                        # [1]
    )
}
```

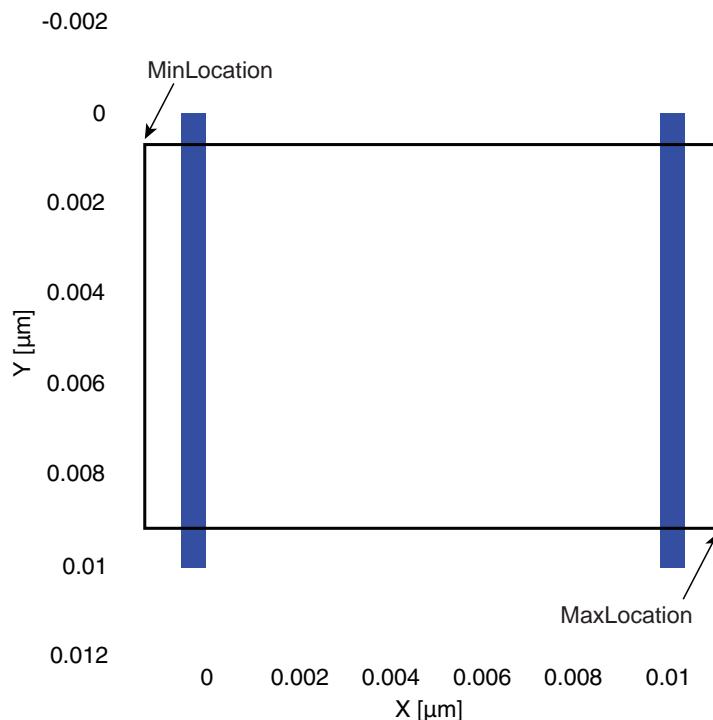
The next sections explain the keywords used in the `KMC` statement.

---

## KMC Simulation Space

You define the portion of the Sentaurus Device structure to be used for the KMC simulations with the keywords `MinLocation` and `MaxLocation`, which define the extremities of a KMC box within the Sentaurus Device structure (see [Figure 75](#)). If part of the specification lies outside of the Sentaurus Device structure, then it is ignored.

*Figure 75 Defining the KMC simulation space in a Sentaurus Device structure*



In addition, with regard to the KMC simulation space:

- It supports both 2D and 3D structures.
- It should include at least one insulator region and at least two electrodes.
- If you use metal regions for contacting insulator regions, then an electrode must contact each metal region. You must also include both the `Poisson` and `Contact` equations when obtaining solutions (see [Performing the Simulation on page 1133](#)).

The following example defines a 10 nm × 10 nm × 10 nm KMC simulation space:

```
Math {
    KMC (
        MinLocation = (0.000, 0.000, 0.000)
        MaxLocation = (0.010, 0.010, 0.010)
```

```
    ReferenceElectrode = "cathode"
)
}
```

For simulations that include the `TDDB` option, defects are confined to KMC cells that span the KMC simulation space (see [Time-Dependent Dielectric Breakdown Simulations on page 1128](#)). You specify the KMC cell size with the `CellLength` keyword (default: 0.0005 μm). When you specify the `TDDB` option, you can specify `HalfCellOffset` to shift the KMC simulation space by a half-cell length relative to the device structure.

The `ReferenceElectrode` keyword specifies the electrode to use as a reference for distances and for some analysis options. If not specified, then the reference electrode defaults to the first electrode that Sentaurus Device finds in the KMC simulation space. After the initial KMC setup, Sentaurus Device automatically changes the reference electrode to be the electrode with the lowest applied bias.

---

## Setting the Minimum Defect Distances

In most cases, defects (traps) are placed randomly in the KMC simulation space. This random placement can cause problems if the defect-to-electrode distance or the defect-to-defect distance is too small:

- The distance might be nonphysical.
- It might result in very high *back-and-forth* defect-to-electrode or defect-to-defect KMC event rates that dominate the KMC simulation time, making it difficult for the simulation to progress to steady state.

To prevent or to help suppress these issues, you can use the keywords `Trap2ElecMinDist` and `Trap2TrapMinDist` to set the minimum-allowed defect-to-electrode distance and defect-to-defect distance, respectively.

The following example sets a minimum distance for these quantities to 1 nm:

```
Trap2ElecMinDist = 0.001
Trap2TrapMinDist = 0.001
```

The default value is 0.0003 μm for both keywords.

---

## Specifying the Random Seed

KMC simulations are stochastic, and each execution of a command file typically produces results that differ to some degree. Specifying a fixed `RandomSeed` greater than zero allows the same results to be obtained from one simulation to the next, assuming the command file and keywords have not changed, and that the simulation is performed on the same platform.

## Omitting KMC Simulations Near Steady State

KMC leakage current simulations are performed during transient ramps in Sentaurus Device. After a Sentaurus Device time step, the KMC simulator is called to account for MIM transport for an interval equal to the Sentaurus Device time step. Control then returns to Sentaurus Device to solve for the next time step, and this cycle repeats until the Sentaurus Device transient `FinalTime` is reached.

With this process, it might be difficult to predict a good `FinalTime` for the Sentaurus Device transient. In many cases, KMC steady-state currents are reached at transient simulation times much smaller than `FinalTime`. If Sentaurus Device continues to call the KMC simulator after this, then the number of KMC events might become excessive, making it difficult to complete the transient.

The `SkipKMCCurrentCheck` keyword allows you to omit further calls to the KMC simulator for the remainder of the Sentaurus Device transient ramp if the MIM total current at two electrodes (or more) changes by less than the specified fraction when compared to the total MIM current from the previous time step, that is, when the simulation is close to steady state.

The following example will cause Sentaurus Device to stop calling the KMC simulator during the transient if the MIM total current change at two electrodes is less than 25% compared to the previous time step:

```
SkipKMCCurrentCheck = 0.25
```

Sometimes, MIM currents might be noisy even when the specified condition is satisfied. Specifying `SkipKMCNumber > 1` can help in this case by forcing Sentaurus Device to satisfy this condition for `SkipKMCNumber` consecutive Sentaurus Device time-steps before omitting KMC.

When the approximate steady-state condition is reached, Sentaurus Device writes the total MIM current and the current due to each separate tunneling process, for all electrodes, to a special MIM current file. See [Special MIM Current File \(Currents and Capacitance\) on page 1139](#).

---

## Omitting the KMC Simulation Completely

For some analysis options, you might need to omit the KMC simulation completely. You can do this by specifying the `SkipKMC` option.

---

## Resetting the Fraction of Defects Filled With Electrons

You can specify the fraction of defects filled with electrons. This fraction is used to fill defects at the start of the first KMC-MIM Transient simulation in the command file (see [Specifying Defects on page 1108](#)). During the transient, the electron configuration might change. By

default, the electron configuration at the end of one `Transient` simulation is used as the initial state for the next `Transient` simulation.

You can modify this behavior by specifying the `ResetElectronFill` option. In this case, the electron-fill configuration established at the start of the first `Transient` in the command file is used at the start of all subsequent `Transient` commands.

---

## Sampling Potential at All Crossings

When calculating the transmission coefficient used in the tunneling rate equations, Sentaurus Device assumes a linear potential between the barrier endpoints. In the case of multilayer structures, a linear potential is assumed between the endpoints of each insulator layer with adjustments of the barrier heights at interfaces to account for electron affinity differences.

In most cases, these assumptions produce sufficiently accurate results. In cases where the potential does not vary linearly across the tunneling barrier (for example, in cylindrical structures), you can obtain improved accuracy by dividing the tunneling path into smaller intervals.

When you specify the `useAllCrossings` option, Sentaurus Device samples the potential for the transmission coefficient calculation at all crossings of the tunneling path with the Sentaurus Device mesh.

---

## Resetting the Rate More Frequently

The KMC simulator calculates the KMC-MIM tunneling rates as needed. After a rate for an event is calculated, it is saved and not recalculated unless Sentaurus Device instructs the KMC simulator to do so.

Sentaurus Device requests a reset of all tunneling rates whenever it encounters a new `Transient` command in the command file, which usually is associated with an electrode bias change and which necessitates updated tunneling rates.

To modify this behavior, select `IncludeMIMChargeInPoisson` in the `KMC_MIM_Transport` statement of the `Physics` section (see [Including the MIM Charge in the Poisson Equation on page 1122](#)). In this case, specifying the option `MoreFrequentRateResets` causes all MIM tunneling rates to be reset after each time step. This can increase the simulation time significantly.

In most cases, resetting the MIM tunneling rates this frequently does not change results significantly.

---

## Adjusting the Calculated Tunneling Rates

The KMC simulator includes algorithms that attempt to adjust high event rates associated with back-and-forth events that might dominate the KMC simulation time and prevent progress toward a steady-state condition. You can use the following keywords to adjust these algorithms:

- `MIMBackAndForthThreshold` sets the number of back-and-forth events that must occur before a rate is adjusted. The default is 2000.
- `MIMEventCountCheckPoint` affects how often the KMC simulator checks for the need to adjust tunneling rates. Checks occur after every `MIMEventCountCheckPoint` event, up to five times. The default is 10000.

---

## Setting the Integration Points

The calculation of some of the tunneling rates used by KMC-MIM simulations requires numeric integration. Sentaurus Device uses a Gauss–Legendre method for these integrations. You use the keyword `IntegrationPoints` to set the number of integration points to use with this method. Larger values result in more accurate but slower simulations. A very conservative value of 256 is used by default. In most cases, 64 to 128 points will provide sufficient accuracy.

---

## Specifying the Approximate Bessel Function

A modified Bessel function of the order  $p$ ,  $I_p(z)$ , is required for the calculation of the following tunneling models:

- `InelasticElectrodeToTrapTunneling` and `InelasticTrapToElectrodeTunneling` (see [Inelastic Electrode-to-Trap and Trap-to-Electrode Tunneling on page 1116](#))
- `TrapToTrapTunneling(MultiPhonon)` (see [Multiphonon Trap-to-Trap Tunneling Model on page 1119](#))

By default, this is obtained from a Math library function. It has been found that, for large numbers of defects, the Math library evaluation can be very slow.

To avoid this slow down, an approximate calculation of  $I_p(z)$  can be used instead by specifying `ApproximateBessel`:

$$I_p(z) \approx \frac{1}{\sqrt{2\pi}} \frac{\exp(\chi)}{\chi^{1/2}} \frac{z^p}{p + \chi} \quad (1214)$$

which is valid for  $\chi \gg 1$ , where  $\chi = \sqrt{p^2 + z^2}$ .

## Specifying Electrodes

Electrodes for KMC-MIM simulations are specified in the usual way using `Electrode` sections in the Sentaurus Device command file.

To define material properties for each electrode that will be used in the KMC-MIM simulation, specify a metal `Material` name in the `Electrode` section to associate with the electrode. The following example associates the properties of the metal material TiN with both the cathode and the anode:

```
Electrode {  
    {name="cathode" voltage=0.0 Material="TiN"}  
    {name="anode"   voltage=0.0 Material="TiN"}  
}
```

The properties of the metal material can then be specified in the parameter file. For example:

```
Material = "TiN" {  
    Bandgap {  
        Workfunction = 4.4      # [eV] Workfunction  
        FermiEnergy = 11.7     # [eV] Fermi energy  
    }  
    KMC_MIM_Transport {  
        metun = 1.0            # [m0] Effective tunneling mass  
        medos = 1.0            # [m0] Density-of-states mass  
    }  
}
```

If the electrode is a semiconductor, then the `KMC_MIM_Transport` section is not needed. The parameters in the `Bandgap` section for the semiconductor are sufficient.

---

## Specifying Insulators

You specify the insulator properties for KMC-MIM simulations in the parameter file. For example:

```
# Associate properties of ZrO2 with "Oxide"  
Material = "Oxide" {  
    Epsilon {  
        epsilon = 40.0      # [1] Permittivity  
    }  
    Bandgap {  
        Chi0 = 2.50       # [eV] Electron affinity,  
                           # conduction band offset (CBO) = 4.4-2.5 = 1.9  
        Eg0  = 5.4        # [eV] Band gap  
    }  
    KMC_MIM_Transport {  
        metun             = 0.50      # [m0] Effective tunneling mass
```

## Chapter 36: Kinetic Monte Carlo MIM Transport

### Specifying Defects

```
medos          = 0.50      # [m0] Density-of-states mass
mcb            = 1.16      # [m0] Conduction band mass
mvb            = 2.50      # [m0] Valence band mass
HuangRhys     = 23        # [1] Huang-Rhys parameter
PhononEnergy   = 0.040    # [eV] Phonon energy
PhononFrequency = 1e13     # [s^-1] Phonon frequency
OpticalPermittivity = 5.6  # [1] Optical permittivity
alpha           = 1.0       # [1] InelasticPhonon parameter
tau0            = 1.e-10    # [s] InelasticPhonon fitting parameter
nu_max          = 1e14      # [s^-1] Maximum frequency for TDDB generation
EA              = 1.87      # [eV] Activation energy for TDDB generation
p0              = 2.1       # [eA] Molecular dipole moment for TDDB generation
SitesPerVolume = 4.4e22    # [cm^-3] Oxygen atoms per volume
SigmaDef        = 10.0     # [S/cm] Conductivity of cells with defects
SigmaIns        = 1e-11    # [S/cm] Conductivity of empty cells
KappaDef        = 0.23     # [W/cm/K] Thermal conductivity of filled cells
KappaIns        = 0.011    # [W/cm/K] Thermal conductivity of empty cells
}
}
```

---

## Specifying Defects

Defects are included in KMC-MIM simulations through specifications in the `Traps` statement of the Sentaurus Device command file. You need to provide a `Traps` statement for each insulator material or region where defects will be placed.

---

## Defining Defects

Defects that are used in KMC-MIM simulations are identified with special trap types that must be included in the `Traps` specification. Different KMC-MIM trap types are allowed that can have different properties: `KmcMim1` and `KmcMim2`. Both trap types can be used in a simulation, but they must be specified in separate `Traps` statements.

The syntax and parameters allowed for `KmcMim1` and `KmcMim2` are shown here:

```
Physics ([Material | Region] = "<insulator_name>") {
    Traps (
        ( [KmcMim1 | KmcMim2]
            EtFill      = <float>                      # [eV]
            EtEmpty     = <float>                      # [eV]
            TrapSigma   = <float>                      # [eV]
            EmptyCharge = <int>                        # [1]
            Conc        = <float>                      # [cm^-3]
            Location    = ((<x1>,<y1>,<z1>) (<x2>,<y2>,<z2>) ...) # [um]
            Xsection    = <float>                      # [cm^2]
            XsectionT2T = <float>                      # [cm^2]
            FillFrac    = <float>                      # [1]
            GrainBoundary(
```

## Chapter 36: Kinetic Monte Carlo MIM Transport

### Specifying Defects

```
        Density=<float>                      # [cm^-2] for 3D device,  
                                              # [cm^-1] for 2D device  
        EtEmpty=<float>                      # [eV]  
        EtFill=<float>                       # [eV]  
        FillFrac=<float>                     # [1]  
        TrapSigma=<float>                     # [eV]  
    )  
)  
}  
}
```

where:

- You specify either `KmcMim1` or `KmcMim2` to define the KMC-MIM trap type.
- `EtFill` sets the filled trap level measured from the conduction band.
- `EtEmpty` sets the empty trap level measured from the conduction band.
- If you specify `TrapSigma`, then trap energy levels are randomized using a Gaussian distribution with this standard deviation.
- `EmptyCharge` sets the charge state of an empty defect (default: 1). This keyword is used only when `IncludeMIMChargeInPoisson` is specified (see [Including the MIM Charge in the Poisson Equation on page 1122](#)).
- `Conc` sets the defect density. The value specified with `Conc` is multiplied by the insulator volume in the KMC simulation space to obtain the number of defects to include in the insulator. For 2D simulations, `AreaFactor` is used for the width of the device in the unsimulated dimension (default: 1  $\mu\text{m}$ ). The calculated number of defects is then randomized over the insulator volume inside the KMC simulation space.
- `Location` sets the locations of specifically placed defects. It allows the locations of defects to be specified instead of, or in addition to, defects that are included using `Conc`.
- If you specify `Xsection`, then this value is used to calculate the defect localization radius used in the inelastic electrode  $\leftrightarrow$  trap rate calculations, using  $r_D = \sqrt{\langle Xsection \rangle / \pi}$ .
- If you specify `XsectionT2T`, then this value is used to calculate the defect localization radius used in the multiphonon trap-to-trap rate calculations, using  $r_D = \sqrt{\langle XsectionT2T \rangle / \pi}$ .
- `FillFrac` sets the initial fraction of defects that are filled with electrons.
- `GrainBoundary` indicates that traps should be generated on the grain boundary. A prerequisite is that the grain field must be generated with `GrainFieldGenerator` specified in the `Math` section:
  - `GrainBoundary(Density=<float>)` specifies the defect density on the grain boundary. The calculated number of defects is then randomized over the insulator volume inside the KMC simulation space.

## Chapter 36: Kinetic Monte Carlo MIM Transport

### Creating the Grain Field

- `GrainBoundary(EtEmpty=<float>)` specifies the empty grain-boundary trap level measured from the conduction band.
- `GrainBoundary(EtFill=<float>)` specifies the filled grain-boundary trap level measured from the conduction band.
- `GrainBoundary(FillFrac=<float>)` specifies the initial fraction of defects that are filled with electrons.
- `GrainBoundary(TrapSigma=<float>)`, if specified, randomizes the grain-boundary trap energy levels by using a Gaussian distribution with this standard deviation.

#### Example

```
Physics (Material = "Oxide") {
    Traps (
        (KmcMim1 Conc=3e18 FillFrac=0.5 EtFill=1.238 EtEmpty=1.283
         GrainBoundary(Density=1e12 FillFrac=0.1 EtFill=1.3
                       EtEmpty=1.2)
        (KmcMim2 Conc=3e18 FillFrac=0.5 EtFill=1.72 EtEmpty=1.7
         TrapSigma=0.02)
    )
}
```

---

## Reading in Defects

Defects are first generated randomly according to the concentration (`Conc`) specified in `Traps` sections. Additional defects can be loaded from TDR files generated by other MIM or time-dependent dielectric breakdown (TDDB) calculations.

To load defects from a TDR file, you can use the keyword `InitialDefects` in the `File` section. The assumption of read-in defects is that the calculation that generated the TDR file uses the same device structure.

```
File {
    InitialDefects = "defects_readin.tdr"      # use Conc = 0.0
}
```

---

## Creating the Grain Field

The traps specified in `GrainBoundary` can be created based on the grain field. To create a grain field, specify a `GrainFieldGenerator` statement in the `Math` section of a particular region:

```
Math(Region=<string>) {
    GrainFieldGenerator (
        [Padding=<float>]                                # units of AverageGrainSize
        [AverageGrainSize=<float>]                         # um
```

## Chapter 36: Kinetic Monte Carlo MIM Transport

### Creating the Grain Field

```
| [AverageGrainSizeVector= (<ax>, <ay>, <az>)] # um
| [NumberOfGrains=<int>] [MinimumGrainSize=<float>] # um
[GrainLocation= ((<x1>,<y1>,<z1>), (<x2>,<y2>,<z2>), ...)] # um
[GrainLocationSize= ((<sx1>,<sy1>,<sz1>),
                     (<sx2>,<sy2>,<sz2>), ...)] # um
[PoissonDistribution]
[RandomSeed=<int>]
[SingleVertexBoundary]
[TwoDimensionalGrains]
[UniformSpatialDistribution]
)
}
```

where:

- Padding, if specified, extends the extremities of the region by a distance of  $\text{Padding} \times a_{gi}$  on all sides before randomizing the grain centers. This allows grain centers to be located outside of the region.
- AverageGrainSize is the approximate average grain size in  $\mu\text{m}$  in  $V_{\text{reg}}$ , or  $V_{\text{box}}$  if padding is used.
- AverageGrainSizeVector allows the grain average size to differ in different directions.
- NumberOfGrains is the number of grain centers located within  $V_{\text{reg}}$ .
- MinimumGrainSize is used to prevent very small grains, which might arise when grain centers are very close to each other. The value specified with this parameter represents the smallest-allowed distance between grain centers.
- GrainLocation sets specific locations where grains are placed in the region. Each grain will be an ellipsoid (or ellipse in two dimensions) with elliptic radii given by GrainLocationSize. The grains specified here will overwrite any of the randomly created grains generated using the previous parameters.
- GrainLocationSize sets the elliptic radii of the grains whose locations are specified with GrainLocation. There must be a one-to-one correspondence between values specified with GrainLocation and GrainLocationSize.
- Specifying PoissonDistribution indicates that the calculated number of grain centers will be used as the expected value for the random number generator of a Poisson distribution. This allows you to account for grain number variability when performing statistical studies.
- RandomSeed is the seed used for the random number generator. Using the same (nonzero) seed in subsequent simulations allows the grain generation to be repeated. The default is zero (new randomization for each simulation).
- Specifying SingleVertexBoundary indicates that only one vertex of edges that cross a Voronoï boundary will be marked with a negative index. This allows grains to be resolved

with a coarser mesh. This is the default. To mark both vertices on an edge, specify -SingleVertexBoundary.

- Specifying TwoDimensionalGrains indicates that grains will be created in two dimensions and extruded in the thinnest region dimension.
- Specifying UniformSpatialDistribution indicates that grain centers will be distributed randomly as previously discussed, by default. However, you might want to have uniformly sized and distributed grains. Specifying this option will attempt to provide this. If you specify this option, then an integral number of grains is distributed uniformly over each dimension of the box volume. In this case, padding is not included:

$$N_x = \max(1, \text{int}((x_{\max} - x_{\min})/a_{gx} + 0.5))$$

$$N_y = \max(1, \text{int}((y_{\max} - y_{\min})/a_{gy} + 0.5))$$

$$N_z = \max(1, \text{int}((z_{\max} - z_{\min})/a_{gz} + 0.5))$$

The total number of grains in the box volume will be given by  $N_{\text{grain}} = N_x N_y N_z$ .

---

## Tunneling Processes

This section discusses tunneling processes and options.

---

### Specifying Tunneling Processes and Related Options

You specify the tunneling processes and related options to include in KMC-MIM simulations in the Physics section of the command file, in the KMC\_MIM\_Transport statement. The available options are listed here and are described in the next sections:

```
Physics {
    Temperature=298
    KMC_MIM_Transport (
        [DirectTunneling]
        [PooleFrenkelEmission]

        [ElasticElectrodeToTrapTunneling]
        [ElasticTrapToElectrodeTunneling]
        [ElasticUsePZeroSZero]

        [InelasticElectrodeToTrapTunneling]
        [InelasticTrapToElectrodeTunneling]
        [InelasticIgnorePMinus]
        [InelasticIncludePZero]
        [InelasticPZeroOnly]

        [DeltaTrapPotential]
```

## Chapter 36: Kinetic Monte Carlo MIM Transport

### Tunneling Processes

```

[TrapToTrapTunneling [(InelasticPhonon) | (MultiPhonon)]]

[ImageChargeBarrierLowering]
[EnergyDependentMass]
[IncludeMIMChargeInPoisson]
[AvgOccupancyInMIMCharge]

[PrintDefectOccupancy]
[PrintMIMEventStatistics]
[PrintMIMRates]

[RateEquationCurrent]
[PrintRECRates]

[ConductivePathCurrent]
[PrintCPCPaths]
[PrintCPCRates]

SensitivityAnalysis (
    Point      = (<xp>, <yp>, <zp>)                      # [um]
    [XtRange   = (<XtStart>, <XtEnd>, <XtStep>)]          # [um]
    [EtRange   = (<EtStart>, <EtEnd>, <EtStep>)]          # [eV]
    [PlotType  = <int>]                                     # [1]
    [PrintAll]
)
[ConductanceEquation]           # Default true (for TDDB only)
[HeatEquation]                  # Default true (for TDDB only)
[TDDB]
)
}
}

```

## Direct Tunneling

You specify the `DirectTunneling` option to select direct tunneling, which accounts for the direct electrode-to-electrode tunneling of electrons.

The KMC direct tunneling rate  $dR_{DT}$  associated with a portion of the source electrode with area  $dA$  is calculated from:

$$dR_{DT} = \frac{dA m_1^* k_B T}{2\pi^2 \hbar^3} \int_0^{E_b} T(E) \ln \left[ \frac{e^{(E_f1 - E_{c1} - E)/k_B T} + 1}{e^{(E_f2 - E_{c1} - E)/k_B T} + 1} \right] dE \quad (1215)$$

where,  $m_1^*$  is the electrode effective tunneling mass (`metun`), and  $T(E)$  is the transmission coefficient calculated using a WKB approximation:

$$T(E) = \exp \left[ -2 \int_{x_1}^{x_2} dx \sqrt{\frac{2m_{ins}^*}{\hbar^2} (V(x) - E)} \right] \quad (1216)$$

## Chapter 36: Kinetic Monte Carlo MIM Transport

### Tunneling Processes

For a semiconductor electrode:

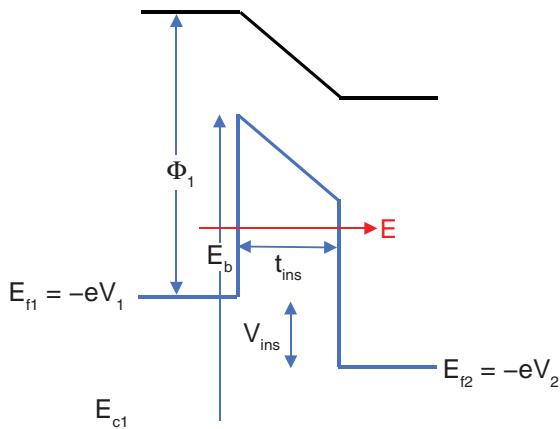
$$dR_{DT} = \frac{dA k_B T}{2\pi^2 \hbar^3} (m_c^* \Theta(E_e - E_c) + m_v^* \Theta(E_v - E_e)) \int_0^{E_b} T(E) \ln \left[ \frac{e^{(E_{f1} - E_{c1} - E)/k_B T} + 1}{e^{(E_{f2} - E_{c1} - E)/k_B T} + 1} \right] dE \quad (1217)$$

If the potential energy  $V(x)$  is linear along the tunneling path, then the integral can be calculated analytically giving:

$$T(E) = \exp \left[ -\frac{4\sqrt{2m_{ins}^*}}{3\hbar(V_{ins}/t_{ins})} [(E_b - E)^{3/2} - (E_b - E - eV_{ins})^{3/2}] \right] \quad (1218)$$

where  $m_{ins}^*$  is the insulator effective tunneling mass ( $\text{metun}$ ).

*Figure 76 Direct tunneling in a MIM structure*



## Poole–Frenkel Emission

You select Poole–Frenkel emission with the `PooleFrenkelEmission` option, which accounts for the field-enhanced thermal emission of electrons from defects to the insulator conduction band.

The KMC Poole–Frenkel emission rate  $R_{PF}$  is given by [1]:

$$R_{PF} = v \cdot \exp \left[ -\frac{E_D}{k_B T} \left( \frac{k_B T}{\beta \sqrt{F}} \right)^2 \left[ 1 + \frac{\beta \sqrt{F}}{k_B T} - 1 \exp \left( \frac{\beta \sqrt{F}}{k_B T} \right) \right] + \frac{1}{2} \right] \quad (1219)$$

where:

- $\beta = \sqrt{\frac{e^3}{\pi \epsilon_0 \epsilon_{\text{opt}}}}$ .

- $F$  is the insulator electric field.
- $\nu$  is the lattice vibration frequency (insulator PhononFrequency).
- $E_D$  is the trap depth (defect EtFill).
- $\epsilon_{\text{opt}}$  is the insulator OpticalPermittivity.

## Elastic Electrode-to-Trap and Trap-to-Electrode Tunneling

You select elastic electrode-to-trap tunneling with the ElasticElectrodeToTrapTunneling option and elastic trap-to-electrode tunneling with the ElasticTrapToElectrodeTunneling option.

The KMC tunneling rates  $R_{\text{ED}}$  and  $R_{\text{DE}}$  are given by [1]:

$$R_{\text{ED}} = \frac{m_e^{5/2} 8 E_e^{3/2}}{m_i^{5/2} 3 \hbar \sqrt{E_D}} f(E_e) T(E_e) \quad (1220)$$

$$R_{\text{DE}} = \frac{m_e^{5/2} 8 E_e^{3/2}}{m_i^{5/2} 3 \hbar \sqrt{E_D}} (1 - f(E_e)) T(E_e) \quad (1221)$$

where:

- $m_e$  is the electrode density-of-states (DOS) mass (medos).
- $m_i$  is the insulator effective tunneling mass (metun).
- $E_D$  is the trap depth of the empty or filled defect (EtEmpty or EtFill).
- $E_e$  is the electron energy.
- $T(E_e)$  is the transmission coefficient for the barrier between the electrode and the defect.
- $f(E_e)$  is the Fermi–Dirac distribution function given by  $f(E_e) = \frac{1}{1 + e^{(E_e - E_f)/(k_B T)}}$ .

For a semiconductor electrode:

$$R_{\text{ED}} = \frac{m_c^{5/2} 8 (E_e - E_c)^{3/2}}{m_i^{5/2} 3 \hbar \sqrt{E_D}} f(E_e) T(E_e) \theta(E_e - E_c) \quad (1222)$$

$$+ \frac{m_v^{5/2} 8 (E_v - E_e)^{3/2}}{m_i^{5/2} 3 \hbar \sqrt{E_D}} f(E_e) T(E_e) \theta(E_v - E_e)$$

$$R_{DE} = \frac{\frac{m_e}{m_i}^{5/2} \frac{8(E_e - E_c)^{3/2}}{3\hbar\sqrt{E_D}} (1 - f(E_e)) T(E_e) \theta(E_e - E_c)}{(1 - f(E_e)) T(E_e) \theta(E_e - E_c)} + \frac{\frac{m_h}{m_i}^{5/2} \frac{8(E_v - E_e)^{3/2}}{3\hbar\sqrt{E_D}} (1 - f(E_e)) T(E_e) \theta(E_v - E_e)}{(1 - f(E_e)) T(E_e) \theta(E_v - E_e)} \quad (1223)$$

## Option for Elastic Tunneling

As an alternative to [Equation 1220](#) and [Equation 1221](#), you can use elastic tunneling rates based on the inelastic expressions in [Inelastic Electrode-to-Trap and Trap-to-Electrode Tunneling](#).

If you specify the `ElasticUsePZeroSZero` option, then the elastic electrode-to-trap and elastic trap-to-electrode tunneling rates are calculated from the  $p = 0$  term in [Equation 1224](#) and [Equation 1225](#) with the Huang–Rhys parameter set to zero ( $S = 0$ ). The latter condition is equivalent to setting  $L_p(z) = 1$  in [Equation 1224](#) and [Equation 1225](#).

## Inelastic Electrode-to-Trap and Trap-to-Electrode Tunneling

You can select inelastic electrode-to-trap tunneling with the option `InelasticElectrodeToTrapTunneling` and inelastic trap-to-electrode tunneling with the `InelasticTrapToElectrodeTunneling` option.

The KMC tunneling rates  $R_{ED}^{\text{MP}}$  and  $R_{DE}^{\text{MP}}$  are given by [1]:

$$R_{ED}^{\text{MP}} = \begin{cases} \infty & p < 0 \\ c_0 N_E(E_p) f(E_p) T(E_p) L_p(z) \cdot e^{\frac{p\hbar\omega}{k_B T}} + \infty & p > 0 \end{cases} \quad (1224)$$

$$R_{DE}^{\text{MP}} = \begin{cases} \infty & p < 0 \\ c_0 N_E(E_p) (1 - f(E_p)) T(E_p) L_p(z) + \infty & p > 0 \\ c_0 N_E(E_p) (1 - f(E_p)) T(E_p) L_p(z) \cdot e^{\frac{-p\hbar\omega}{k_B T}} & p > 0 \end{cases} \quad (1225)$$

where:

- The electron tunneling energy  $E_p$  is given by  $E_p = E_e + p\hbar\omega$ , where  $p$  is the integer phonon number, and  $\hbar\omega$  is the phonon energy in the insulator (`PhononEnergy`).
- The DOS in the electrode  $N_E(E_p)$  is given by:

$$N_E(E_p) = \frac{1}{2\pi^2 \hbar^2} \frac{2m_e^{3/2}}{\sqrt{E_p - E_c}} \theta(E_p - E_c) \quad (1226)$$

where  $m_e$  is the electrode DOS mass (`medos`), and  $E_c$  is the conduction band edge of the electrode.

## Chapter 36: Kinetic Monte Carlo MIM Transport

### Tunneling Processes

- The DOS in the semiconductor electrode  $N_E(E_p)$  is given by:

$$N_E(E_p) = \frac{1}{2\pi^2} \frac{(2m_c)^{3/2}}{\hbar^2} \sqrt{E_p - E_c} \cdot \theta(E_p - E_c) + \frac{1}{2\pi^2} \frac{(2m_v)^{3/2}}{\hbar^2} \sqrt{E_v - E_p} \cdot \theta(E_v - E_p) \quad (1227)$$

where  $m_c$ ,  $m_v$  is the electrode DOS mass (medos), and  $E_c$  the conduction band edge of the electrode.

- The capture coefficient  $c_0$  is given by:

$$c_0 = \frac{(4\pi)^2 r_D^3}{\hbar E_g} (\hbar E_0)^3 \quad (1228)$$

$$(\hbar E_0)^3 = \frac{e^2 \hbar^2 F^2}{2m_i} \quad (1229)$$

where:

- $E_g$  is the insulator band gap (Eg0).
- $m_i$  is the insulator effective tunneling mass (metun).
- $F$  is the insulator electric field.
- The localization radius  $r_D$  is given by  $r_D = \frac{\hbar}{\sqrt{2m_i E_D}}$ .

If you specify DeltaTrapPotential, then a field-independent  $c_0$  is used instead:

$$c_0 = \frac{(4\pi)^2 r_D^3}{\hbar} (E_D)^2 \quad (1230)$$

- The Fermi–Dirac distribution  $f(E_p)$  is  $f(E_p) = \frac{1}{1 + e^{(E_p - E_f)/(k_B T)}}$ .
- The multiphonon transition probability  $L_p(z)$  is given by:

$$L_p(z) = \exp\left[-S(2f_{\text{BE}} + 1) + \frac{p\hbar\omega}{2k_B T}\right] \cdot I_p(z) \quad (1231)$$

where:

- $S$  is the Huang–Rhys parameter in the insulator (HuangRhys).
- $f_{\text{BE}}$  is the Bose–Einstein distribution:

$$f_{\text{BE}} = \frac{1}{\exp[\frac{\hbar\omega}{k_B T}] - 1} \quad (1232)$$

- $I_p(z)$  is a modified Bessel function of order  $p$  with:

$$z = 2S\sqrt{f_{\text{BE}}(f_{\text{BE}} + 1)} \quad (1233)$$

## Options for Inelastic Tunneling

You can use variants of these expressions, mainly related to which phonon energies are included in the tunneling rate calculations. The following options allow the phonon energy summations in [Equation 1224](#) and [Equation 1225](#) to be modified:

- `InelasticIgnorePMinus`: Ignores the  $p < 0$  terms in [Equation 1224](#) and [Equation 1225](#)
- `InelasticIncludePZero`: Includes the  $p = 0$  term in [Equation 1224](#) and [Equation 1225](#)
- `InelasticPZeroOnly`: Uses only the  $p = 0$  term in [Equation 1224](#) and [Equation 1225](#)

## Trap-to-Trap Tunneling

This section discusses the trap-to-trap tunneling models.

### Trap-to-Trap Tunneling Model

You select trap-to-trap tunneling (both elastic and inelastic) with the `TrapToTrapTunneling` option, which accounts for electron tunneling between defects. The trap-to-trap rate  $R_{ij}$  from defect  $i$  to defect  $j$  is given by [1]:

$$R_{ij} = v \cdot T_{ij} \cdot \begin{cases} \exp \left( -\frac{\Delta E_{ji}}{k_B T} \right), & \text{if } \Delta E_{ji} > 0 \\ 1, & \text{otherwise} \end{cases} \quad (1234)$$

where:

- $v$  is the lattice vibration frequency (insulator `PhononFrequency`).
- $T_{ij}$  is the transmission coefficient for the energy barrier between defect  $i$  and defect  $j$ .
- $\Delta E_{ji}$  is the energy difference between the final defect state and the initial defect state.

This model is an extension of the Miller–Abrahams hopping model [1] and is obtained by replacing  $\exp(-2r_{ij}/r_D)$  in the Miller–Abrahams expression with a calculation of the transmission coefficient  $T_{ij}$ .

### Inelastic Phonon Trap-to-Trap Tunneling Model

You can select the inelastic phonon trap-to-trap tunneling model as an alternative to the `TrapToTrapTunneling` model by specifying `TrapToTrapTunneling(InelasticPhonon)`.

When selected, the trap-to-trap electron capture rate from defect 1 to defect 2 is calculated using:

$$R_{12} = \frac{1}{\tau_0} \cdot T_{12}(E) \cdot \left[ \frac{\alpha(S-p)^2}{S} + 1 - \alpha \right] \cdot L_p(z) \cdot \begin{cases} \exp \left( -\frac{\Delta E_{21}}{k_B T} \right), & \text{if } \Delta E_{21} > 0 \\ 1, & \text{otherwise} \end{cases} \quad (1235)$$

where:

- $\tau_0$  is a fitting parameter (insulator tau0; default =  $10^{-4}$  s).
- $T_{12}(E)$  is the transmission coefficient for the energy barrier between defect 1 and defect 2.
- $\alpha$  is a user-specified parameter (insulator alpha; default = 1).
- $S$  is the Huang–Rhys parameter (insulator HuangRhys).
- $p$  is the phonon number given by:

$$p = \left| \frac{E_1 - E_2}{\hbar\omega} \right| \quad (1236)$$

where  $\hbar\omega$  is the phonon energy in the insulator (PhononEnergy).

- $\Delta E_{21}$  is the energy difference between the trap levels of defect 2 and defect 1.
- $L_p(z)$  is the multiphonon transition probability (see [Equation 1231](#) to [Equation 1233](#)).

The phonon number,  $p$ , is usually taken to be an integer. In the implementation of this model, however,  $p$  can take a floating-point value given by [Equation 1236](#), and the modified Bessel function,  $I_p(z)$  in [Equation 1231](#), is approximated with the expression given in [Specifying the Approximate Bessel Function on page 1106](#).

## Multiphonon Trap-to-Trap Tunneling Model

You can select a multiphonon description of trap-to-trap tunneling [2][3] by specifying TrapToTrapTunneling(MultiPhonon). For a transition from defect 1 to defect 2, this model provides expressions for capture and emission.

For capture by defect 2 from defect 1, the transition rate is given by:

$$R_{ca, 12} = c_0 N_1(E_{2,p}) f_1(E_{2,p}) T_{12}(E_{2,p}) L_p(z) \quad (1237)$$

$p = 0$

$$N_1(E_{2,p}) = \frac{1}{2\pi^2} \frac{(2m)^{3/2}}{\hbar^2} \sqrt{E_{2,p} - E_1} \cdot \Theta(E_{2,p} - E_1) \quad (1238)$$

## Chapter 36: Kinetic Monte Carlo MIM Transport

### Tunneling Processes

$$f_1(E_{2,p}) = \frac{1}{1 + e^{(E_{2,p} - E_1)/(k_B T)}} \quad (1239)$$

$$E_{2,p} = E_2 + p\hbar\omega \quad (1240)$$

For emission from defect 1 to defect 2, the transition rate is given by:

$$R_{\text{em}, 12} = \int_{p=0}^{\infty} c_0 N_2(E_{1,p})(1 - f_2(E_{1,p})) T_{12}(E_{1,p}) L_p(z) e^{-\frac{p\hbar\omega}{k_B T}} \quad (1241)$$

$$N_2(E_{1,p}) = \frac{1}{2\pi^2} \frac{(2m_2)^{3/2}}{\hbar^2} \sqrt{E_{1,p} - E_2} \cdot \Theta(E_{1,p} - E_2) \quad (1242)$$

$$f_2(E_{1,p}) = \frac{1}{1 + e^{(E_{1,p} - E_2)/(k_B T)}} \quad (1243)$$

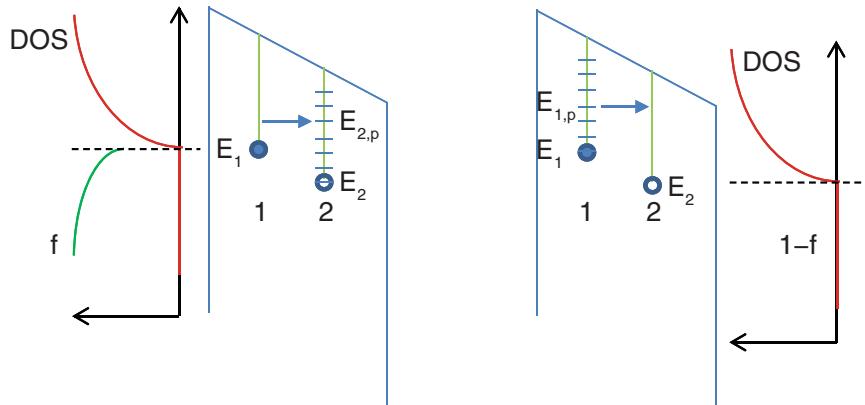
$$E_{1,p} = E_1 + p\hbar\omega \quad (1244)$$

In [Equation 1237](#) and [Equation 1241](#),  $c_0$  is taken from [Equation 1228](#) ( $c_0$  is taken from [Equation 1230](#) if with the DeltaTrapPotential option),  $L_p(z)$  is taken from [Equation 1231](#), and  $T_{12}(E)$  is the transmission coefficient calculated at energy  $E$ . In [Equation 1238](#) and [Equation 1242](#),  $m_1$  and  $m_2$  are DOS masses (<sub>medos</sub>).

For the KMC trap-to-trap rate, the minimum of  $R_{\text{ca}, 12}$  and  $R_{\text{em}, 12}$  is used:

$$R_{12} = \min(R_{\text{ca}, 12}, R_{\text{em}, 12}) \quad (1245)$$

*Figure 77 (Left) Capture and (right) emission between two defects*



## Image Charge Barrier Lowering

To account for image charge barrier lowering, specify the ImageChargeBarrierLowering option. When selected, this effect is included in all the preceding tunneling rate calculations

## Chapter 36: Kinetic Monte Carlo MIM Transport

### Tunneling Processes

except for Poole–Frenkel emission and trap-to-trap tunneling. Image charge barrier lowering reduces the transmission coefficient barrier along the tunneling path by:

$$\Delta V_{\text{ICBL}} = -\frac{e^2}{16\pi\epsilon_0\epsilon_{\text{opt}}x} \quad (1246)$$

where:

- $x$  is the distance from the electrode.
- $\epsilon_{\text{opt}}$  is the insulator optical permittivity (Optical Permittivity).

#### Note:

Including [Equation 1246](#) in the calculation of the transmission coefficient introduces a nonlinearity requiring a numeric integration. Often, you can account for image charge barrier lowering by simply reducing the conduction band offset (the workfunction of the electrode minus the electron affinity of the insulator) by a small amount (such as 0.15 eV), thereby avoiding the extra numeric integration.

## Energy-Dependent Mass

You can account for the energy-dependent tunneling mass by specifying `EnergyDependentMass`. When selected, this effect is included in all the preceding tunneling rate calculations except for Poole–Frenkel emission. The energy-dependent mass calculation adjusts the effective tunneling mass in the top half of insulator band gap using the following expression:

$$m_{\text{ins}}^*(E') = \begin{cases} \frac{m_c}{m_v} \frac{1 - \frac{E'}{E_g}}{1 - \frac{m_c}{m_v} \frac{E'}{E_g}}, & E' < \frac{E_g}{2} \\ \frac{m_c m_v}{m_c + m_v}, & E' \geq \frac{E_g}{2} \end{cases} \quad (1247)$$

where:

- $E'$  is the energy difference between the insulator conduction band and the electron tunneling energy.
- $E_g$  is the insulator band gap ( $Eg0$ ).
- $m_c$  is the insulator conduction band mass ( $mcb$ ).
- $m_v$  is the insulator valence band mass ( $mvb$ ).

**Note:**

Including [Equation 1247](#) in the calculation of the transmission coefficient introduces a nonlinearity requiring a numeric integration. Often, you can account for the effect of energy-dependent mass by simply adjusting the insulator effective tunneling mass, thereby avoiding the extra numeric integration.

---

## Including the MIM Charge in the Poisson Equation

By default, the charge associated with defects and electrons in the insulator are excluded when solving the Poisson equation. Therefore, the effect this charge has on the insulator potential is ignored when calculating tunneling rates.

To include the defect and electron charge in the solution of the Poisson equation, select the `IncludeMIMChargeInPoisson` option. When selected, the following adjustments to the solution process are invoked:

- At the start of each Sentaurus Device time step, the spatial distribution of charge due to defects and electrons is updated, by assigning the charge associated with each defect and each electron in the insulator to the nearest Sentaurus Device vertex. The charge concentration at each vertex is computed by dividing the net vertex charge by the vertex volume.
- The updated charge distribution is included in the solution of the Poisson equation to obtain a new potential distribution.
- If this is the first time step of a `Transient` ramp, then Sentaurus Device instructs the KMC simulator to reset all tunneling rates. The KMC simulator discards all previously saved tunneling rates and requests updated rates from Sentaurus Device based on the updated potential. By default, the rates established with this potential are used for all subsequent time steps during the transient, even though the calculated potential in the insulator might change due to movement of electrons.
- If you select the `MoreFrequentRateResets` option, then the tunneling rates are updated after every Sentaurus Device time step.

Note the following:

- The charge associated with a defect is established by specifying `EmptyCharge` in the `Traps` statement associated with the defect (see [Specifying Defects on page 1108](#)).
- Resetting the saved KMC tunneling rates at every Sentaurus Device time step (by selecting `MoreFrequentRateResets`) can significantly increase the simulation time and, in most cases, is not needed.
- Including the defect and electron charge in the solution of the Poisson equation usually only causes a small perturbation of the insulator potential, especially at low defect

concentrations and high biases. In most cases, sufficient accuracy is obtained by excluding the `IncludeMIMChargeInPoisson` option.

## Average Electron Occupancy in the MIM Charge

By default, when using `IncludeMIMChargeInPoisson`, the electron component of the total MIM charge at each defect is taken to be the actual occupancy at the end of the KMC time interval, which is zero (0) or one (1). In most cases, it will be zero because the average occupancy associated with a defect tends to be small.

As an alternative, the average occupancy associated with a defect can be used for the electron charge by specifying the option `AvgOccupancyInMIMCharge`. The average occupancies of the defects are calculated from a solution of the rate equations for the system of defects (see [Rate Equation Current Calculation on page 1123](#)) and the selected tunneling models.

---

## Printing Options

You can use the following options to print additional output:

- `PrintDefectOccupancy` prints a defect summary, including occupancy and energy, to the log file and to standard output (see [Occupancy and Energy on page 1136](#)).
  - `PrintMIMEventStatistics` prints MIM event statistics to standard output (see [Event Statistics on page 1137](#)).
  - `PrintMIMRates` prints the calculated tunneling rates to the log file and to standard output (see [Tunneling Rates on page 1138](#)).
- 

## Rate Equation Current Calculation

In addition to determining MIM leakage current using the KMC simulator, the option `RateEquationCurrent` invokes a solution of the rate equations for the system of defects and selected tunneling models. This capability, which is used by default, provides average occupancy for all defects and the total MIM leakage current.

The equation for the occupancy of the  $i^{\text{th}}$  defect in a system of  $N_d$  defects and  $N_e$  electrodes is given by:

$$\begin{array}{ccc} N_d & & N_e \\ [f_j(1-f_i)R_{ji}-f_i(1-f_j)R_{ij}] + (1-f_i) & R_{ki}-f_i & R_{ik} = 0 \\ j \neq 1 & k=1 & k=1 \end{array} \quad (1248)$$

where:

- $R_{ij}$  and  $R_{ji}$  are the total tunneling rates between defect  $i$  and defect  $j$ .
- $R_{ik}$  and  $R_{ki}$  are the total tunneling rates between defect  $i$  and electrode  $k$ .

This leads to an  $N_d \times N_d$  system of equations, which is solved to obtain the defect occupancies.

From the occupancies, the leakage current at the  $k^{\text{th}}$  electrode is then computed from:

$$I_k = q \sum_{i=1}^{N_d} [f_i R_{ik} - (1-f_i) R_{ki}] \quad (1249)$$

A summary of each solution of the rate equations is written to the `*.log` file. For example:

Rate Equation Solution					
Iter	Rhs	max_error	TrapIndex	Occupancy	Update
0	1.6386e-02				
1	6.2237e-12	9.1199e+07	2	3.6728e-13	-4.2615e-12
2	9.4081e-19	6.5440e-01	7	8.0304e-07	-5.2551e-14
... converged solution					
Electrode	Voltage(V)	Current(A)			
cathode	0.0000e+00	3.4146e-20			
anode	-2.2000e+00	-3.4146e-20			

The total current calculated by the rate equation current (REC) method is written to the `*_mimcur_des.plt` file and can be selected for plotting in Sentaurus Visual using `MIMRateEquationCurrent`.

You can specify the option `PrintRECRates` to print the tunneling rates that are calculated during a REC solution.

The setup for the REC method is the same as that for a KMC-MIM leakage current calculation. The REC method uses the same defects, models, parameters, and biasing as used in a KMC calculation. To perform REC solutions without running a KMC simulation, specify `SkipKMC` in the `KMC` statement of the `Math` section.

## Conductive Path Current Analysis

Conductive path current (CPC) analysis provides an efficient method for approximating the leakage current in MIM structures. CPC analysis attempts to find favorable conduction paths through the defects in the MIM structure, with the assumption that each defect can belong to only one path. The current through each path is calculated, and the total leakage current is obtained by summing the current from each path.

You specify CPC analysis with the `ConductivePathCurrent` option.

You can specify the option `PrintCPCPaths` to obtain detailed information about the paths used in the CPC calculation as well as the option `PrintCPCRates` to obtain information about the rate calculations used during the CPC calculation.

The total current calculated by CPC analysis is written to the `*_mimcur_des.plt` file and can be selected for plotting in Sentaurus Visual using `MIMConductivePathCurrent`.

The setup for CPC analysis is the same as that for a KMC-MIM leakage current calculation. CPC analysis uses the same defects, models, parameters, and biasing as used in a KMC calculation. To perform a CPC analysis without running a KMC simulation, specify `SkipKMC` in the `KMC` statement of the `Math` section.

---

## Sensitivity Analysis

Sensitivity analysis can determine efficiently how the position and the energy level of a single defect will affect the leakage current in a MIM structure.

A single defect is placed at different locations,  $X_t$ , along a path through the structure and a leakage current calculation is performed at a series of defect energy levels,  $E_t$ , at each location. The result is a matrix of leakage current values, one for each  $(X_t, E_t)$  pair. The maximum leakage current and the corresponding  $(X_t, E_t)$  are printed to the log file.

The leakage current for all  $(X_t, E_t)$  pairs is written to a file with the `_<volt>_mimsa_des.tdr` suffix, which Sentaurus Visual can read for plotting leakage current over the  $(X_t, E_t)$  space, where `<volt>` is the voltage at which the analysis occurs.

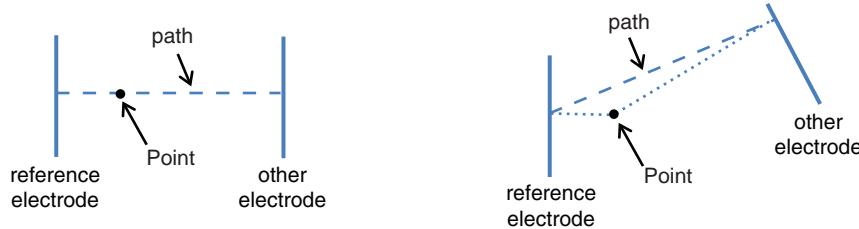
Alternatively, the leakage current can be plotted as absolute energy versus  $X_t$ , which shows the band edges. The base name for the file can be specified in the `File` section of the command file using the `MIMSensitivity` keyword. In this file,  $X_t$  values are stored in nm and  $E_t$  values are stored in eV.

You specify sensitivity analysis in the `KMC_MIM_Transport` statement of the `Physics` section as follows:

```
Physics {
    KMC_MIM_Transport (
        ...
        SensitivityAnalysis (
            Point      = (<xp>, <yp>, <zp>)                                # [um]
            [XtRange  = (<XtStart>, <XtEnd>, <XtStep>)]                      # [um]
            [EtRange  = (<EtStart>, <EtEnd>, <EtStep>)]                      # [eV]
            [PlotType = <int>]                                                 # [1]
            [PrintAll]
        )
        ...
    )
}
```

where:

- `Point` specifies the path for the defect locations,  $X_t$ . Sentaurus Device finds the closest point on the reference electrode to `Point` and the closest point on another electrode. The path is drawn between these two closest electrode points. In most cases, `Point` will be on the path (as shown in the left figure). In some situations, this might not be the case (as shown in the right figure). The default is `Point= (0, 0, 0)`.



- `XtRange` specifies the start, end, and step size for defect locations,  $X_t$ , along the path. The reference electrode is at  $X_t = 0$ . Defect locations,  $X_t$ , cannot be closer than 0.0001  $\mu\text{m}$  to the electrodes, and the minimum-allowed step size is 0.0001  $\mu\text{m}$ . If you do not specify `XtRange`, then  $X_t$  will span the entire path, from electrode to electrode, with a step size of 0.0001  $\mu\text{m}$ .
- `EtRange` specifies the start, end, and step size for defect energy levels,  $E_t$ , relative to the conduction band. The defect energy levels cannot be closer than 0.05 eV to the band edges, and the minimum-allowed step size is 0.05 eV. If you do not specify `EtRange`, then  $E_t$  will span the entire band gap with a step size of 0.05 eV.
- `PlotType` specifies how Sentaurus Visual plots the results. Options are:
  - `PlotType=1`: (Default) Plot absolute energy versus  $X_t$  (flat band).
  - `PlotType=2`: Plot absolute energy versus  $X_t$  (bias included).
  - `PlotType=3`: Plot  $E_t$  versus  $X_t$ .
- If you specify `PrintAll`, then the leakage current values for all  $(X_t, E_t)$  pairs are printed to the log file.

The setup for sensitivity analysis is the same as that for a KMC-MIM leakage current calculation. That is, a `Traps` statement is required that specifies either `KmcMim1` or `KmcMim2`, and a series of `Quasistationary` or `Transient` ramps should be specified to set electrode biases and to initiate the analysis. To perform sensitivity analysis without running KMC, specify `SkipKMC` in the `KMC` statement of the `Math` section.

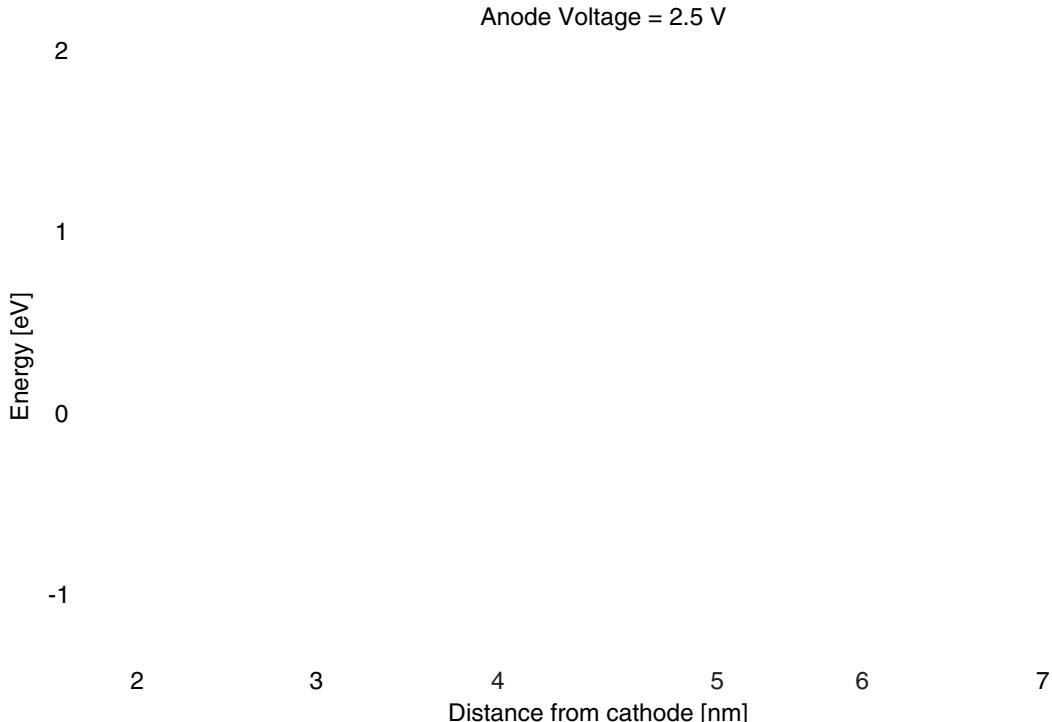
## Chapter 36: Kinetic Monte Carlo MIM Transport Tunneling Processes

Here is an example log file and graphical output for a two-layer MIM structure (see [Figure 78](#)):

```
Sensitivity Analysis
Path Start Location: ( 0.0000e+00, 5.0000e-03, 5.0000e-03) um, cathode
Path End Location : ( 1.0000e-02, 5.0000e-03, 5.0000e-03) um, anode
Path Max Distance : 1.0000e-02 um
Defect Distances   : Start: 1.0000e-04 um, End: 9.9000e-03 um,
                      Step: 1.0000e-04 um
Defect Energies    : Start: 5.0000e-02 eV, End: 5.3500e+00 eV,
                      Step: 5.0000e-02 eV

Maximum current      = 1.2670e-17 Amps
Defect energy level = 1.2000e+00 eV
Distance along path = 2.0000e-03 um
Defect Location     = ( 2.0000e-03, 5.0000e-03, 5.0000e-03) um
```

*Figure 78 Example of graphical output (PlotType=1) from sensitivity analysis*



## Time-Dependent Dielectric Breakdown Simulations

If you select the `TDDB` option, then Sentaurus Device accounts for time-dependent dielectric breakdown (TDDB). The KMC simulator is used to generate defects according to the following expression [4]:

$$G(x, y, z) = v \cdot \exp \frac{E_A - p_0 \frac{2 + \kappa}{3} \cdot F(x, y, z)}{k_B T(x, y, z)} \quad (1250)$$

where:

- $v$  is the maximum frequency of the generation process (`nu_max`); default:  $10^{14} \text{ s}^{-1}$ .
- $E_A$  is the activation energy (`EA`); default: 2.4 eV ( $\text{SiO}_2$ ) [4], 3.0 eV ( $\text{HfO}_2$ ) [4].
- $p_0$  is the molecular dipole moment (`p0`); default: 4.1 eÅ ( $\text{SiO}_2$ ) [4], 5.2 eÅ ( $\text{HfO}_2$ ) [4].
- $\kappa$  is the dielectric constant.
- $F$  is the local electric field.
- $T$  is the local lattice temperature.

Parameter values for  $\text{ZrO}_2$  from [5][6] are  $E_A = 1.87 \text{ eV}$  and  $p_0 = 2.1 \text{ eÅ}$ . Parameter values for  $\text{SiO}_2$  from [7] considering a double electron capture process are  $E_A = 0.7 \text{ eV}$  and  $p_0 = 1.0 \text{ eÅ}$ .

When you specify `TDDB`, Sentaurus Device automatically uses the following settings:

- `SkipKMC` (KMC statement in the `Math` section), which instructs Sentaurus Device not to use the KMC simulator for the calculation of leakage current
- `RateEquationCurrent` (KMC\_MIM\_Transport statement in the `Physics` section), which instructs Sentaurus Device to calculate the current through the MIM structure using the REC method

The other parameters in [Tunneling Processes](#) define the tunneling mechanisms and options used for leakage current calculations. The MIM leakage current is calculated after each Sentaurus Device/KMC time interval and is stored in the standard `*.plt` file as `MIMTotalCurrent`.

In a nonuniform insulator, there can be a spatial distribution on the activation energy and molecule dipole moment. To simulate the nonuniformity, assuming that the spatial variation of the activation energy and molecule dipole moment follow a normal distribution, their standard deviations  $\sigma_{E_A}$  (`SigmaEA`) and  $\sigma_{p_0}$  (`SigmaP0`) can be defined. By default, both values are zero, meaning there is no spatial variation on the activation energy and molecule dipole moment.

The conductive effect of individual defects should be independent of the size of the simulation mesh. To ensure that the conductive effect has an influence across multiple meshes, the conductive radius (`ConductiveRadius`) can be defined in units of  $\mu\text{m}$ . Meshes within this radius will have a high conductance. By default, a conductive radius is 0 means the defect does not influence neighboring meshes.

When you specify `TDDB`, the `ConductanceEquation` and `HeatEquation` options are switched on by default. The steady-state conductance equation is given by:

$$\vec{\nabla} \cdot \vec{j} = \vec{\nabla} \cdot \sigma \vec{\nabla} \psi = 0 \quad (1251)$$

The steady-state heat equation with Joule heating is given by:

$$\vec{\nabla} \cdot \kappa \vec{\nabla} T + \psi \sigma \vec{\nabla} \psi = 0 \quad (1252)$$

In these equations,  $\psi$  is the potential,  $T$  is the temperature, and  $\sigma$  and  $\kappa$  are the spatially dependent electrical and thermal conductivities that account for the presence of defects in KMC cells.

The solution of these equations over the KMC cells provides better values for the potential (and electric field) and temperature to use in the generation expression, [Equation 1250](#). If you select `-HeatEquation`, then the heat equation is not solved. If you select `-ConductanceEquation`, then neither equation is solved.

## Adjustment for Vacancy Sites per Volume

[Equation 1250](#) is an expression for the vacancy generation rate at a single site. In KMC simulations, since only one vacancy is allowed to occupy a KMC cell, a rate adjustment must be made to account for the cell size used in the simulation.

To this end, [Equation 1250](#) is multiplied by a factor that represents the number of vacancies that could potentially occupy the volume of a KMC cell:

$$\text{factor} = \text{SitesPerVolume} \times \text{CellLength}^3 \frac{10^{-4} \text{ cm}^3}{\mu\text{m}^3} \quad (\text{in three dimensions}) \quad (1253)$$

or:

$$\begin{aligned} \text{factor} &= \text{SitesPerVolume} \times \text{CellLength}^2 \quad (\text{in two dimensions}) \\ &\times \text{AreaFactor} \frac{10^{-4} \text{ cm}^3}{\mu\text{m}^2} \end{aligned} \quad (1254)$$

where `SitesPerVolume` is an insulator property that is specified in the parameter file. The default is 4.40e22 ( $\text{SiO}_2$ ), 5.54e22 ( $\text{HfO}_2$ ), and 5.55e22 ( $\text{ZrO}_2$ ) in units of  $\#/cm^3$ .

## TDDB Break Criteria

Specifying the `UseKMCTimeStep` option instructs Sentaurus Device to use the time step determined by the KMC simulator. The keyword `MaxTrapNumberIncrease` allows you to specify the maximum increment of the number of particles of each KMC time step. In addition, you can use `MaxCurrent` and `MaxCurrentRatio` as stopping criteria for the TDDB calculation. These keywords and option are specified in the `KMC` statement of the `Math` section. For example:

```
Math {
    Extrapolate
    transient=BE
    KMC (
        ...
        UseKMCTimeStep
        MaxTrapNumberIncrease = 25
        MaxTrapNumber = 1000
        MaxCurrent = 1e-6
        MaxCurrentRatio = 1e8
    )
}
```

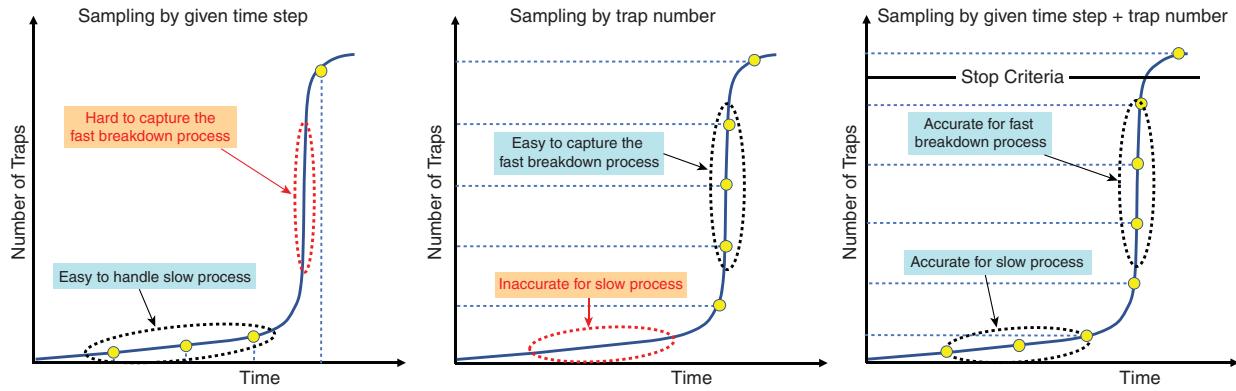
where:

- `UseKMCTimeStep` is false if not provided. The KMC simulator determines the time step for the transient calculation using the value of `MaxTrapNumberIncrease` when the trap increases sufficiently fast.
- `MaxTrapNumberIncrease` is 0 by default. Each time step allows a maximum number of particles to increase if `UseKMCTimeStep` is specified.
- `MaxTrapNumber` is 0 by default. Use the value specified as the maximum trap number as a criterion to stop the TDDB calculation.
- `MaxCurrent` is 0 by default. Use the value specified as the maximum current as a criterion to stop the TDDB calculation.
- `MaxCurrentRatio` is 0 by default. Use the value specified as the maximum current ratio as a criterion to stop the TDDB calculation.

The current ratio is defined as  $I(t)/\min(I(t') \text{ for } t' < t)$ .

The current stopping criteria work without `UseKMCTimeStep` and `MaxTrapNumberIncrease`, but the criteria will work better when both of these are provided because more data points are sampled during the breakdown process.

**Figure 79** Different ways of sampling: (left) sampling by given time step, (middle) sampling by trap number increment, and (right) sampling by given time step and trap number



## TDDB Models

Different TDDB generation models are suggested by Wu [8]:

$$G = G_0 \exp \left[ a \frac{|E|^b}{|E_0|^b} \right] \frac{|E|^c}{|E_0|^c} \quad (1255)$$

where:

- $G_0 = v \exp - \frac{E_a - p_0(2 + \kappa)|E|/3}{k_B T}$
- $G_0$  is the model implemented as the default TDDB generation model.
- $a, b, c$ , and  $E_0$  are parameters for the new TDDB models.
- $p$  and  $E_0$  are defined in the \*.par file. For example:

```
Material = "ZrO2" {
    KMC_MIM_Transport {
        TrapGenParam1 = 1.1, 0.5, 0.0, 0.0
            # parameters a, b, and c in Equation 1255
            # for kmcmim1
            # default Parameter_1 = 0.0, 0.5, 0.0
        FieldRef1 = 1.0e6 # parameter E0 for kmcmim1 (V/cm)
        TrapGenParam2 = 2.1, 0.5, 0.0, 0.0
            # parameters a, b and c in Equation 1255
            # for kmcmim2
            # default Parameter_2 = 0.0, 0.5, 0.0
        FieldRef2 = 1.0e6 # parameter E0 for kmcmim2 (V/cm)
    }
}
```

where:

- For the default model, these parameters do not need to be specified.
- For the inverse E model, use `TrapGenParam1= a, -1.0, 0.0, 0.0` with  $a < 0$ .
- For the square root of the E model, use `TrapGenParam1= a, 0.5, 0.0, 0.0` with  $a > 0$ .
- For the power law model, use `TrapGenParam1= 0.0, 0.0, c, 0.0` with  $c > 0$ .
- The fourth parameter in `TrapGenParam` is related to the fluence dependency (see [Fluence Dependency](#)).

## Fluence Dependency

The fluence of an electrode  $F(t)$  and the fluence density at location  $x$  ( $f(x, t)$ ) are defined by:

$$\begin{aligned} F(t) &= \int_0^t I(\tau) d\tau \\ f(x, t) &= \int_0^t j(x, \tau) d\tau \end{aligned} \quad (1256)$$

where  $I(t)$  is the current flowing through the electrode, and  $j(x, t)$  is the current density at location  $x$ .

Experiments show that the trap density versus fluence follows the square root law  $N_T(t) \propto F(t)^m$ , with close to 0.5 [9]. Therefore, the total generation rate follows:

$$G = \frac{dN_T(t)}{dt} \propto \frac{dF(t)^m}{dt} = F(t)^{m-1} I(t) \quad (1257)$$

For the generation rate at a given location, you add the fluence-dependent term to the existing electric field-dependent model  $G_E(x)$ :

$$G_f(x, t) = G_E(x) \frac{f(x, t)^{m-1} j(x, t)}{f_0^m} \quad (1258)$$

Here,  $f_0$  is the user-defined reference fluence.

In the command file, no new option is introduced for the fluence model. You must include `MoreFrequentRateResets` in the `KMC` statement of the `Math` section to update the generation rate at every time step, because fluence is time dependent.

In the parameter file, no new option is introduced for the fluence model. You must include `MoreFrequentRateResets` in the `KMC` statement of the `Math` section to update the generation rate at every time step, because fluence is time dependent.

## Chapter 36: Kinetic Monte Carlo MIM Transport

### Performing the Simulation

In the parameter file:

```
Material = "ZrO2" {
    KMC_MIM_Transport {
        FluenceRef1 = 1e5      # [um^-2]; reference fluence for defect 1
        FluenceRef2 = 1e5      # [um^-2]; reference fluence for defect 2
        TrapGenParam1 = 0,0,0,0.5  # Fourth parameter is the m
        TrapGenParam2 = 0,0,0,0.5  # Fourth parameter is the m
    }
}
```

The fourth parameter in `TrapGenParam` is the parameter exponent  $m$  for the fluence power law dependency.

---

## Performing the Simulation

KMC-MIM simulations and KMC-TDDB simulations are performed during Transient ramps specified in the `Solve` section of the command file. During a transient ramp, Sentaurus Device first obtains a solution for a time step,  $\Delta t$ , during which the state of KMC is frozen. After this, the KMC simulator is called for an equal time interval,  $\Delta t$ , where:

- For MIM leakage current simulations, the KMC simulator accounts for the transport events contributing to leakage current, after which MIM electrode currents are updated.
- For TDDB simulations, the KMC simulator accounts for defect generation during the interval, after which the steady-state current through the MIM structure is calculated using the REC method.

Then, control returns to Sentaurus Device and a solution is obtained for the next time step. This sequence continues until the Sentaurus Device transient ramp has finished or, in the case of a MIM leakage current simulation, until steady state is reached.

In most cases, the KMC simulation space will consist only of insulators and electrodes, so it is usually only necessary to include the `Poisson` equation in the Sentaurus Device solutions. If metal regions are included in the structure, then also include the `Contact` equation in the solution.

You can use Quasistationary solutions to set electrodes to the required bias. You can include multiple Quasistationary or Transient sequences in the `Solve` section to perform the simulations at different biases.

For example, for a KMC-MIM leakage current simulation:

```
Solve {
    Coupled {Poisson Contact}
    NewCurrentFile="tran_25C_"

    # Ramp anode to 2 V
    Quasistationary (
```

## Chapter 36: Kinetic Monte Carlo MIM Transport

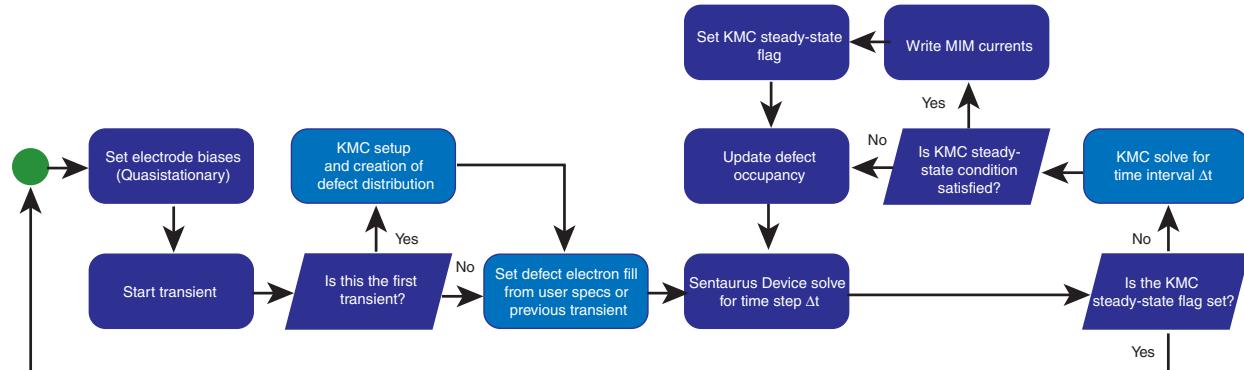
### Printed Output

```
InitialStep=0.5 MinStep=1e-3 MaxStep=0.5
Goal {Name="anode" Voltage=2.0}
) { Coupled {Poisson Contact} }
Transient (
    InitialTime=0 FinalTime=1000 Increment=2 InitialStep=1e-10
    MaxStep=1e30
) { Coupled {Poisson Contact} }

# Ramp anode to 2.5 V
Quasistationary (
    InitialStep=0.51 MinStep=1e-3 MaxStep=0.5
    Goal {Name="anode" Voltage=2.5}
) { Coupled {Poisson Contact} }
Transient (
    InitialTime=0 FinalTime=1000 Increment=2 InitialStep=1e-10
    MaxStep=1e30
) { Coupled {Poisson Contact} }
}
```

For a KMC-TDDB simulation, these Quasistationary or Transient sequences are similar, except a smaller Increment is usually chosen to try to better capture the current increases caused by defect generation.

Figure 80 Typical Sentaurus Device/KMC flow for MIM leakage current simulations



## Printed Output

This section discusses the various printed output from KMC-MIM or KMC-TDDB simulations.

## Setup Information

By default, at the start of a transient ramp that includes KMC-MIM or KMC-TDDB, the setup information is printed to the log file and to standard output.

## Chapter 36: Kinetic Monte Carlo MIM Transport

### Printed Output

For example:

```
KMC setup information:
  MinLocation[0] =  0.000e+00, MaxLocation[0] =  9.000e-03
  MinLocation[1] =  0.000e+00, MaxLocation[1] =  4.000e-02
  MinLocation[2] =  0.000e+00, MaxLocation[2] =  4.000e-02
Regions in KMC simulation space for MIM:
  Oxide: Insulator
  ReferenceElectrode: cathode
    Maximum shortest distance to anode =  9.0000e-03 (um)
    Minimum shortest distance to anode =  9.0000e-03 (um)
  RandomSeed: 130988961
Tunneling mechanisms and settings:
  DirectTunneling
  PooleFrenkelEmission
  TrapToTrapTunneling
  InelasticElectrodeToTrapTunneling
  InelasticTrapToElectrodeTunneling
  InelasticIncludePZero
  InelasticIgnorePMinus
  ImageChargeBarrierLowering
  EnergyDependentMass
  ResetElectronFill

Type 1 defects: total = 43
index dist2ref(um) x(um) y(um) z(um)          Efill(eV) Eempt(eV) EmptyChrg region
  1 1.2211e-03 ( 1.2211e-03, 1.0409e-03, 3.0509e-02) 1.1500 1.1500 1.00 Oxide
  2 1.2381e-03 ( 1.2381e-03, 1.7439e-02, 1.5901e-02) 1.1500 1.1500 1.00 Oxide
  3 1.2498e-03 ( 1.2498e-03, 2.0123e-02, 2.1410e-02) 1.1500 1.1500 1.00 Oxide
  4 1.3742e-03 ( 1.3742e-03, 3.6005e-02, 3.6985e-02) 1.1500 1.1500 1.00 Oxide
  5 1.4255e-03 ( 1.4255e-03, 1.8202e-02, 2.5212e-03) 1.1500 1.1500 1.00 Oxide
  6 1.5850e-03 ( 1.5850e-03, 2.5704e-02, 2.1066e-02) 1.1500 1.1500 1.00 Oxide
  7 1.7802e-03 ( 1.7802e-03, 3.1963e-02, 5.5743e-03) 1.1500 1.1500 1.00 Oxide
  8 1.8567e-03 ( 1.8567e-03, 9.1129e-03, 3.0628e-02) 1.1500 1.1500 1.00 Oxide
  9 2.0098e-03 ( 2.0098e-03, 2.0942e-02, 2.6813e-02) 1.1500 1.1500 1.00 Oxide
 10 2.1568e-03 ( 2.1568e-03, 1.0286e-02, 1.8557e-02) 1.1500 1.1500 1.00 Oxide
 11 2.1743e-03 ( 2.1743e-03, 1.4167e-02, 5.3378e-03) 1.1500 1.1500 1.00 Oxide
 12 2.3214e-03 ( 2.3214e-03, 2.7644e-03, 4.3473e-03) 1.1500 1.1500 1.00 Oxide
 13 2.3973e-03 ( 2.3973e-03, 3.8677e-02, 2.6157e-02) 1.1500 1.1500 1.00 Oxide
 14 2.4480e-03 ( 2.4480e-03, 2.8581e-02, 2.2959e-02) 1.1500 1.1500 1.00 Oxide
 15 2.5130e-03 ( 2.5130e-03, 3.5078e-02, 2.7776e-02) 1.1500 1.1500 1.00 Oxide
 16 2.6195e-03 ( 2.6195e-03, 2.3512e-02, 2.6541e-02) 1.1500 1.1500 1.00 Oxide
 17 2.7903e-03 ( 2.7903e-03, 4.5697e-03, 2.5779e-03) 1.1500 1.1500 1.00 Oxide
 18 2.8706e-03 ( 2.8706e-03, 3.4315e-02, 2.2018e-02) 1.1500 1.1500 1.00 Oxide
 19 2.9317e-03 ( 2.9317e-03, 6.6882e-03, 2.4940e-02) 1.1500 1.1500 1.00 Oxide
 20 3.0575e-03 ( 3.0575e-03, 3.7761e-02, 2.6641e-02) 1.1500 1.1500 1.00 Oxide
 21 3.2296e-03 ( 3.2296e-03, 3.7586e-02, 3.2086e-02) 1.1500 1.1500 1.00 Oxide
...
...
MIM defect and electron summary:
  Defects : total = 43
  Defect1 : total = 43
  Defect2 : total = 0
```

## Chapter 36: Kinetic Monte Carlo MIM Transport

### Printed Output

```
Electrons: total = 0
Region : Oxide
Defects : 43
Defect1 : 43
```

At the end of each Sentaurus Device time step, a MIM current update is given. For example:

```
MIM electron count and current at electrodes:
Electrode      TotalCount   tStepCount Component    Current (Amps)
cathode        -202          -103  Total Curr  -3.0018e-19
cathode         0             0     Direct Tun  0.0000e+00
cathode         0             0     PF emiss   0.0000e+00
cathode        -13169        -2891 Inel El2Tr  -8.4254e-18
cathode        12967          2788  Inel Tr2El  8.1253e-18

anode          202           103  Total Curr  3.0018e-19
anode          0             0     Direct Tun  0.0000e+00
anode          202           103  PF emiss   3.0018e-19
anode          0             0     Inel El2Tr  0.0000e+00
anode          0             0     Inel Tr2El  0.0000e+00
```

---

## Occupancy and Energy

Specify `PrintDefectOccupancy` in the `KMC_MIM_Transport` statement of the `Physics` section to print defect location, energy, and occupancy to the log file and to standard output after each Sentaurus Device time step. For example:

```
Type 1 defects: total = 20
index dist2ref(um)      x(um)      y(um)      z(um)      E(eV)      KMCOcc
RECOccup Charge(e)      region
  1  5.6754e-04  ( 5.6754e-04, 7.0437e-03, 3.5963e-03)  1.1000  0
2.2779e-10  0.0000e+00  oxide
  2  1.1722e-03  ( 1.1722e-03, 8.6942e-03, 1.1224e-02)  1.1000  0
2.9579e-06  0.0000e+00  oxide
  3  1.9155e-03  ( 1.9155e-03, 1.3907e-02, 6.0403e-03)  1.1000  0
4.4543e-03  0.0000e+00  oxide
  4  2.1378e-03  ( 2.1378e-03, 1.0529e-03, 4.6743e-03)  1.1000  0
6.4094e-03  0.0000e+00  oxide
  5  2.4049e-03  ( 2.4049e-03, 5.1541e-03, 2.9232e-03)  1.1000  0
5.9990e-03  0.0000e+00  oxide
  6  2.6702e-03  ( 2.6702e-03, 8.6097e-04, 3.5008e-03)  1.1000  0
4.7246e-03  0.0000e+00  oxide
  7  3.2154e-03  ( 3.2154e-03, 8.3628e-03, 1.3953e-02)  1.1000  0
1.6267e-03  0.0000e+00  oxide
  8  3.3788e-03  ( 3.3788e-03, 1.8238e-04, 9.7925e-03)  1.1000  0
1.0865e-03  0.0000e+00  oxide
  9  3.4087e-03  ( 3.4087e-03, 5.6450e-03, 4.6252e-03)  1.1000  0
1.0162e-03  0.0000e+00  oxide
10  3.4438e-03  ( 3.4438e-03, 1.2257e-02, 6.4376e-03)  1.1000  0
9.2199e-04  0.0000e+00  oxide
11  3.8271e-03  ( 3.8271e-03, 1.4823e-02, 8.8857e-03)  1.1000  0
2.2940e-04  0.0000e+00  oxide
12  3.8570e-03  ( 3.8570e-03, 1.5582e-02, 1.3997e-02)  1.1000  0
1.9144e-04  0.0000e+00  oxide
```

## Chapter 36: Kinetic Monte Carlo MIM Transport

### Printed Output

```
13 4.2468e-03 ( 4.2468e-03, 1.3063e-02, 3.7237e-03) 1.1000 0
4.8615e-06 0.0000e+00 oxide
14 4.2479e-03 ( 4.2479e-03, 6.2982e-03, 1.5123e-02) 1.1000 0
4.8923e-06 0.0000e+00 oxide
15 5.3886e-03 ( 5.3886e-03, 1.5756e-02, 1.9473e-03) 1.1000 0
2.3180e-09 0.0000e+00 oxide
...
...
```

---

## Event Statistics

Specify `PrintMIMEventStatistics` in the `KMC_MIM_Transport` statement of the `Physics` section to print MIM event statistics to standard output after each Sentaurus Device time step. For example:

```
----- Direct Tunneling Event Statistics (Total events= 0) -----
----- Poole-Frenkel Emission Event Statistics (Total events= 10) -----
    3 Trap1 15:( 0.00251299, 0.0350785, 0.0277761) (um) to anode
    3 Trap1 20:( 0.0030575, 0.0377605, 0.0266408) (um) to anode
    1 Trap1 16:( 0.00261949, 0.0235122, 0.0265409) (um) to anode
    1 Trap1 10:( 0.00215679, 0.010286, 0.0185566) (um) to anode
    1 Trap1 17:( 0.0027903, 0.00456968, 0.00257792) (um) to anode
    1 Trap1 14:( 0.00244797, 0.0285811, 0.0229586) (um) to anode
----- Inelastic Electrode-to-Trap Tunneling Event Statistics (Total
events= 2823) -----
    609 cathode to Trap1 1:( 0.00122109, 0.00104095, 0.030509) (um)
    577 cathode to Trap1 2:( 0.00123813, 0.0174387, 0.0159013) (um)
    527 cathode to Trap1 3:( 0.00124977, 0.0201226, 0.0214097) (um)
    371 cathode to Trap1 4:( 0.00137417, 0.0360052, 0.0369845) (um)
    344 cathode to Trap1 5:( 0.00142546, 0.0182019, 0.00252116) (um)
    171 cathode to Trap1 6:( 0.00158498, 0.0257042, 0.0210655) (um)
    76 cathode to Trap1 7:( 0.00178023, 0.0319631, 0.00557431) (um)
    65 cathode to Trap1 8:( 0.00185673, 0.00911292, 0.0306282) (um)
    33 cathode to Trap1 9:( 0.0020098, 0.0209418, 0.0268126) (um)
    17 cathode to Trap1 10:( 0.00215679, 0.010286, 0.0185566) (um)
    17 cathode to Trap1 11:( 0.00217431, 0.0141669, 0.00533778) (um)
    5 cathode to Trap1 12:( 0.00232142, 0.00276436, 0.00434726) (um)
    4 cathode to Trap1 15:( 0.00251299, 0.0350785, 0.0277761) (um)
    3 cathode to Trap1 13:( 0.00239733, 0.0386775, 0.0261574) (um)
    3 cathode to Trap1 14:( 0.00244797, 0.0285811, 0.0229586) (um)
    1 cathode to Trap1 16:( 0.00261949, 0.0235122, 0.0265409) (um)
----- Inelastic Trap-to-Electrode Tunneling Event Statistics (Total
events= 2813) -----
    609 Trap1 1:( 0.00122109, 0.00104095, 0.030509) (um) to cathode
    577 Trap1 2:( 0.00123813, 0.0174387, 0.0159013) (um) to cathode
    527 Trap1 3:( 0.00124977, 0.0201226, 0.0214097) (um) to cathode
    371 Trap1 4:( 0.00137417, 0.0360052, 0.0369845) (um) to cathode
    344 Trap1 5:( 0.00142546, 0.0182019, 0.00252116) (um) to cathode
    171 Trap1 6:( 0.00158498, 0.0257042, 0.0210655) (um) to cathode
    76 Trap1 7:( 0.00178023, 0.0319631, 0.00557431) (um) to cathode
    65 Trap1 8:( 0.00185673, 0.00911292, 0.0306282) (um) to cathode
    33 Trap1 9:( 0.0020098, 0.0209418, 0.0268126) (um) to cathode
```

## Chapter 36: Kinetic Monte Carlo MIM Transport

### Quantities Available for Plotting

```
17 Trap1 11:( 0.00217431, 0.0141669, 0.00533778) (um) to cathode
16 Trap1 10:( 0.00215679, 0.010286, 0.0185566) (um) to cathode
  4 Trap1 12:( 0.00232142, 0.00276436, 0.00434726) (um) to cathode
  2 Trap1 14:( 0.00244797, 0.0285811, 0.0229586) (um) to cathode
  1 Trap1 15:( 0.00251299, 0.0350785, 0.0277761) (um) to cathode
----- Trap-to-Trap Tunneling Event Statistics (Total events= 7784) -----
 3893 Trap1 13:( 0.002397, 0.038677, 0.026157) (um) to Trap1
  20:( 0.003057, 0.037760, 0.026640) (um)
 3890 Trap1 20:( 0.003057, 0.037760, 0.026640) (um) to Trap1
  13:( 0.002397, 0.038677, 0.026157) (um)
  1 Trap1 12:( 0.002321, 0.002764, 0.004347) (um) to Trap1
  17:( 0.002790, 0.004569, 0.002577) (um)

...
...
```

---

## Tunneling Rates

Specify `PrintMIMRates` in the `KMC_MIM_Transport` statement of the `Physics` section to print calculated tunneling rates to the log file and to standard output. For example:

```
Tunneling rates calculated during current KMC time interval:
Tunneling Process      From           To          Distance(um) Rate(#/s)
  Direct tunneling     (cathode -to- anode      ) 9.0000e-03  6.5571e-09
  Inelastic elec-to-trap (cathode -to- TrapIndex 29) 5.8662e-03  5.8716e-20
  Inelastic elec-to-trap (anode   -to- TrapIndex 29) 3.1338e-03  3.0745e-21
  Inelastic elec-to-trap (cathode -to- TrapIndex 37) 7.1969e-03  1.6854e-28
  Inelastic elec-to-trap (anode   -to- TrapIndex 37) 1.8031e-03  1.0023e-11
  Inelastic elec-to-trap (cathode -to- TrapIndex 16) 2.6195e-03  2.4392e-01
  Inelastic elec-to-trap (anode   -to- TrapIndex 16) 6.3805e-03  8.5563e-38
  Inelastic elec-to-trap (cathode -to- TrapIndex 39) 7.2884e-03  4.2489e-29
  Inelastic elec-to-trap (anode   -to- TrapIndex 39) 1.7116e-03  4.9441e-11
  Inelastic elec-to-trap (cathode -to- TrapIndex 43) 7.9966e-03  9.0767e-34
  Inelastic elec-to-trap (anode   -to- TrapIndex 43) 1.0034e-03  1.6124e-05
  Inelastic elec-to-trap (cathode -to- TrapIndex 42) 7.6953e-03  9.0120e-32
  Inelastic elec-to-trap (anode   -to- TrapIndex 42) 1.3047e-03  6.8110e-08
  Inelastic elec-to-trap (cathode -to- TrapIndex 35) 6.8916e-03  1.6095e-26
  Inelastic elec-to-trap (anode   -to- TrapIndex 35) 2.1084e-03  5.2010e-14
...
...
```

---

## Quantities Available for Plotting

This section presents the quantities that can be plotted.

---

## Quantities Written to the Current Plot (\*.plt) File

After each Sentaurus Device time step, the following quantities are written to the `*.plt` file for each electrode in the KMC simulation space.

## Chapter 36: Kinetic Monte Carlo MIM Transport

### Quantities Available for Plotting

These can be visualized in Sentaurus Visual:

```
MIMConductionCurrent  
MIMDirectTunnelingCurrent  
MIMElasElec2TrapCurrent  
MIMElasTrap2ElecCurrent  
MIMInelElec2TrapCurrent  
MIMInelTrap2ElecCurrent  
MIMLeakageCurrent  
MIMPooleFrenkelCurrent  
MIMTotalCurrent
```

---

## Special MIM Current File (Currents and Capacitance)

Sentaurus Device creates a special file containing the approximate steady-state MIM currents. KMC leakage currents calculated by Sentaurus Device are written to this file only when the `SkipKMCCurrentCheck` condition is satisfied. If the simulation command file includes transient ramps for several bias points, then the data in this file includes the approximate steady-state MIM currents for each bias. This makes it convenient to create plots of the approximate steady-state current versus bias.

In addition, Sentaurus Device calculates the `AverageCharge`,  $Q_{avg}$ , on each electrode during the transient ramp up to the time where the `SkipKMCCurrentCheck` condition is satisfied, which is also written to the MIM current file. For each pair of consecutive biases, a quantity called `MIMCapacitance` is also included, which is calculated from:

$$C(V_2) = \frac{Q_{avg}(V_2) - Q_{avg}(V_1)}{V_2 - V_1} \quad (1259)$$

By default, the name of the MIM current file is the base name of the command file with the suffix `_mimcur_des.plt`. Alternatively, you can specify the base name of the file using the `MIMCurrent` keyword in the `File` section of the command file.

The following example creates a file named `MyResults_mimcur_des.plt`:

```
File {  
    ...  
    MIMCurrent = "MyResults"  
    ...  
}
```

---

## Quantities Written to TDR Files

The following section discuss the quantities written to TDR files.

## Concentration Plots

You can specify the following quantities in the `Plot` section of the command file:

- `MIMDefect1Conc`: Concentration of `KmcMim1` defects in the Sentaurus Device mesh
- `MIMDefect2Conc`: Concentration of `KmcMim2` defects in the Sentaurus Device mesh
- `MIMDefectConc`: Total defect concentration in the Sentaurus Device mesh
- `MIMElectronConc`: Electron concentration in the Sentaurus Device mesh
- `MIMChargeConc`: Net charge concentration (defects and electrons) in the Sentaurus Device mesh
- `MIMCellPotential`: KMC cell potential from the conductance equation (`TDDB` only)
- `MIMCellTemperature`: KMC cell temperature from the heat equation (`TDDB` only)

These quantities provide an approximate representation of where defects and electrons are located. They are obtained by first assigning each defect or electron to the closest vertex in the Sentaurus Device mesh. The concentration is then calculated by dividing the number of defects or electrons at the vertex by the volume associated with the vertex.

## Particle Plots

You can specify particle plots of `KmcMim1` defects, `KmcMim2` defects, and electron locations in the `Plot` section of the command file with the `MIMParticle` plot name. The particle written to a TDR file can be loaded as the initial defect distribution in other MIM or TDDB calculations.

Particle plots show the actual position of the MIM defects within the Sentaurus Device structure when visualized with Sentaurus Visual.

When viewing the TDR file in Sentaurus Visual, you can do the following:

- Select the particles to be displayed on the **Lines/Particles** tab of the Selection panel.
- Modify the size and color of the particles by choosing **Data > Region Properties**.
- Select the quantities `Par(Current)`, `Par(Energy)`, and `Par(Occupancy)` to display the particles with colors that represent their current, energy level, and electron occupancy, respectively.

---

## Band Diagram With Traps

When a KMC-MIM simulation is performed, Sentaurus Device automatically creates a TDR file containing information that allows plotting of a band diagram that will show the defect distance from a reference electrode and defect energy within the insulator band gap.

## Chapter 36: Kinetic Monte Carlo MIM Transport

Example: MIM Leakage Current

You specify the reference electrode with the `ReferenceElectrode` keyword in the `KMC` statement of the `Math` section.

By default, the name of the MIM band diagram file is the base name of the command file with the suffix `_mimband_des.tdr`. Alternatively, you can specify the base name of the file using the `MIMBand` keyword in the `File` section of the command file.

The following example creates a file named `MyResults_mimband_des.tdr`:

```
File {  
    ...  
    MIMBand = "MyResults"  
    ...  
}
```

To plot the band diagram:

1. Open the TDR file with Sentaurus Visual.
2. In the Selection panel, select the data name `<basename>_mimband_des_BandDiagram`.
3. Set the following:
  - a. Select **X** and click **To X-Axis**.
  - b. Select **Ec** and click **To Left Y-Axis**.
  - c. Select **Ev** and click **To Left Y-Axis**.
4. Select the data name `<basename>_mimband_des_TrapLevel`.
5. Set the following:
  - a. Select **TrapPos** and click **To X-Axis**.
  - b. Select **TrapE** and click **To Left Y-Axis**.
6. Select the **TrapE** curve and change its curve properties in the Plot Properties panel, on the **Main** tab, by selecting **Curve Markers** and clearing **Curve Lines** if already selected.

---

## Example: MIM Leakage Current

This example demonstrates a KMC-MIM simulation of the leakage current in a TiN/ZrO<sub>2</sub>/TiN MIM structure. The ZrO<sub>2</sub> film is 9 nm thick.

## Chapter 36: Kinetic Monte Carlo MIM Transport

Example: MIM Leakage Current

---

### Sentaurus Device Command File

```
# mim_9nm.cmd
File {
    Grid      = "mim9x40x40s.tdr"
    Plot      = "mim_9nm"
    Current   = "mim_9nm"
    Output    = "mim_9nm"
    Param     = "mim.par"
    MIMCurrent = "mim_9nm"
    MIMBand   = "mim_9nm"
}
Plot {
    VertexIndex
    ConductionBandEnergy
    ValenceBandEnergy
    InsulatorElectricField
    Potential
    SpaceCharge
    MIMDefectConc
    MIMElectronConc
    MIMChargeConc
    MIMParticleDefect1
    MIMParticleDefect2
    MIMParticleElectron
}
Electrode {
    {name="cathode" voltage=0.0 Material="TiN"}
    {name="anode"   voltage=0.0 Material="TiN"}
}
Physics {
    Temperature=298
    KMC_MIM_Transport (
        ImageChargeBarrierLowering
        EnergyDependentMass
        #IncludeMIMChargeInPoisson
        DirectTunneling
        PooleFrenkelEmission
        TrapToTrapTunneling
        #ElasticElectrodeToTrapTunneling
        #ElasticTrapToElectrodeTunneling
        #ElasticUsePZeroSZero
        InelasticElectrodeToTrapTunneling
        InelasticTrapToElectrodeTunneling
        InelasticIncludePZero
        InelasticIgnorePMinus

        PrintMimRates
        PrintMIMEventStatistics
        #PrintDefectOccupancy
    )
}
```

## Chapter 36: Kinetic Monte Carlo MIM Transport

Example: MIM Leakage Current

```
Physics (Material = "Oxide") {
    Traps (
        (KmcMim1 Conc=3e18 FillFrac=0.0 EtFill=1.15 EtEmpty=1.15
         EmptyCharge=1)
    )
}
Math {
    Extrapolate
    NumberOfThreads=1
    Transient=BE
    KMC (
        MinLocation = (0.00 , 0.00, 0.00)
        MaxLocation = (0.009, 0.04, 0.04)
        Trap2ElecMinDist = 0.0010
        Trap2TrapMinDist = 0.0010
        RandomSeed=130988961

        SkipKMCCurrentCheck = 0.10
        SkipKMCNumber = 1
        #SkipKMC

        ResetElectronFill
        #UseAllCrossings
        #MoreFrequentRateResets
        IntegrationPoints = 128
    )
}
Solve {
    Coupled {Poisson}
    NewCurrentFile="tran_25C_"

    Quasistationary (
        InitialStep=0.1 MinStep=1e-3 MaxStep=0.5
        Goal {Name="anode" Voltage=0.75}
    ) { Coupled {Poisson} }
    Transient (
        InitialTime=0 FinalTime=1e5 Increment=2 InitialStep=1e-10
        MaxStep=1e30
    ) { Coupled {Poisson Contact} }

    Quasistationary (
        InitialStep=0.1 MinStep=1e-3 MaxStep=0.5
        Goal {Name="anode" Voltage=1.0}
    ) { Coupled {Poisson} }
    Transient (
        InitialTime=0 FinalTime=1e5 Increment=2 InitialStep=1e-10
        MaxStep=1e30
    ) { Coupled {Poisson Contact} }
}
...
}
```

## Chapter 36: Kinetic Monte Carlo MIM Transport

Example: MIM Leakage Current

---

### Sentaurus Device Parameter File

```
# mim.par
Material = "TiN" {
    Bandgap {
        WorkFunction = 4.7          # [eV]
        FermiEnergy  = 11.7         # [eV]
    }
    KMC_MIM_Transport {
        metun = 1.00                # [m0]
        medos = 1.00                # [m0]
    }
}
Material = "Oxide" {
    KMC_MIM_Transport {
        metun      = 0.85          # [m0]
        medos      = 1.00          # [m0]
        mcb        = 1.16          # [m0]
        mvb        = 2.5           # [m0]
        HuangRhys = 1.0            # [1]
        PhononEnergy = 0.040       # [eV]
        OpticalPermittivity = 5.6 # [1]
        PhononFrequency = 1e13     # [s^-1]
        alpha       = 1             # [1]
        tau0       = 1.e-4          # [s]
    }
    Epsilon {
        # Use permittivity of ZrO2
        epsilon = 40.0            # [1]
    }
    Bandgap {
        # Specify values for ZrO2
        Chi0 = 2.96               # [eV] CBO = 4.7-2.96 = 1.74
        Eg0  = 5.4                 # [eV]
    }
}
```

---

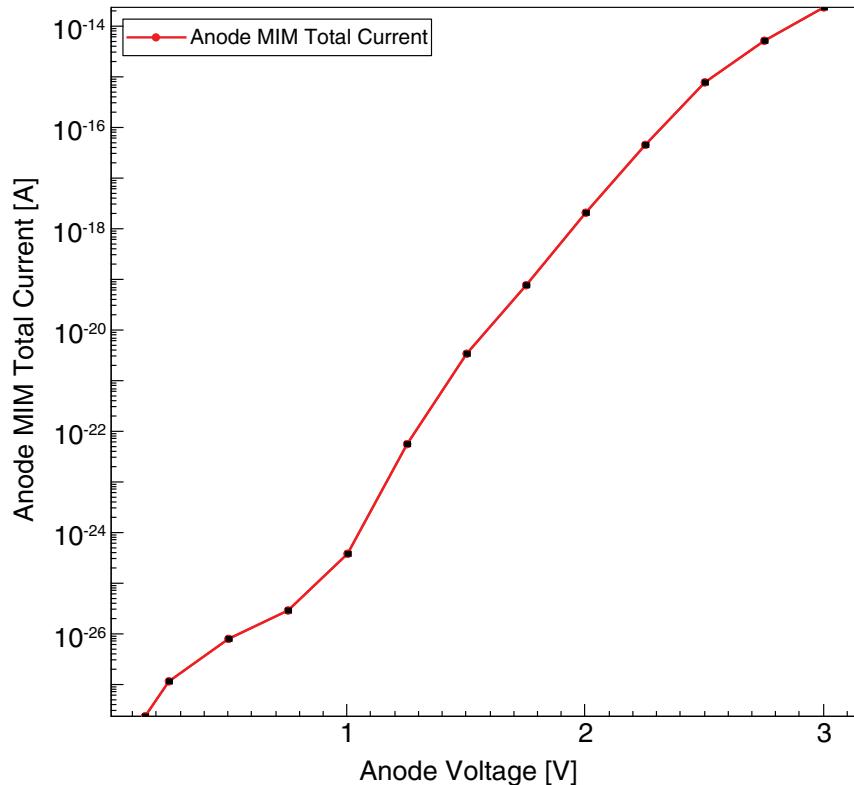
### Results

[Figure 81](#) shows the plot that was produced using the results written to the file `mim_9nm_mimcur_des.plt`.

## Chapter 36: Kinetic Monte Carlo MIM Transport

Example: TiN/ZrO<sub>2</sub>/TiN KMC-TDDB Simulation

Figure 81 Approximate steady-state leakage current in a KMC-MIM simulation



---

## Example: TiN/ZrO<sub>2</sub>/TiN KMC-TDDB Simulation

### Note:

This example was created before the introduction of the `HeatEquation` and `ConductanceEquation` options. Using these options will affect the results significantly.

This example demonstrates a KMC-TDDB simulation for a TiN/ZrO<sub>2</sub>/TiN MIM structure. The ZrO<sub>2</sub> film is 8 nm thick.

This example examines whether the initial TDDB simulation capabilities, despite lacking some key features, can generate results that are qualitatively similar to those shown in Figure 4 of [6].

## Chapter 36: Kinetic Monte Carlo MIM Transport

Example: TiN/ZrO<sub>2</sub>/TiN KMC-TDDB Simulation

This example uses the following defect and TDDB generation parameters. The generation-related parameters have been modified compared to those given in [6] so as to have similar breakdown times:

- $T = 398$  K
- $t_{\text{ins}} = 8$  nm
- CBO = 1.90 eV: ( $\text{WF}_{\text{TiN}} = 4.4$  eV) – ( $\chi_{\text{ZrO}_2} = 2.5$  eV)
- $E_T = 1.1$  eV
- $N_T = 10^{19}$  cm<sup>-3</sup>
- $E_A = 3.0$  eV
- $p_0 = 2.05$  eÅ
- $\kappa_{\text{ZrO}_2} = 40$
- $\text{nu\_max} = 1.5 \times 10^{13}$  s<sup>-1</sup>
- $\text{SitesPerVolume} = 5.55 \times 10^{22}$  cm<sup>-3</sup>

---

## Sentaurus Device Command File

```
# tddb_zro2.cmd
File {
    Grid      = "zro2_8x16x16.tdr"
    Plot      = "tddb_zro2"
    Current   = "tddb_zro2"
    Output    = "tddb_zro2"
    MIMCurrent = "tddb_zro2"
    MIMBand   = "tddb_zro2"
    Param     = "tddb.par"
}
Plot {
    VertexIndex
    ConductionBandEnergy
    ValenceBandEnergy
    InsulatorElectricField
    LatticeTemperature
    Potential
    MIMDefect1Conc
    MIMDefect2Conc
    MIMDefectConc
    MIMElectronConc
    MIMChargeConc
    MIMParticle
}
Electrode {
```

## Chapter 36: Kinetic Monte Carlo MIM Transport

Example: TiN/ZrO<sub>2</sub>/TiN KMC-TDBB Simulation

```
{name="cathode" voltage=0.0 Material="TiN" }
{name="anode"   voltage=0.0 Material="TiN" }
}
Physics {
    Temperature=398
    KMC_MIM_Transport (
        TDBB
        #IncludeMIMChargeInPoisson

        ConductivePathCurrent
        PrintCPCPaths

        DirectTunneling
        PooleFrenkelEmission
        TrapToTrapTunneling(MultiPhonon)

        InelasticElectrodeToTrapTunneling
        InelasticTrapToElectrodeTunneling
        InelasticIncludePZero
        InelasticIgnorePMinus
    )
}
Physics (Material = "ZrO2") {
    Traps (
        (KmcMim1 Conc=1e19 FillFrac=0.0 EtFill=1.1 EtEmpty=1.1
         TrapSigma=0 EmptyCharge=0 Xsection=1e-14 XsectionT2T=1e-14)
    )
}
Math {
    Extrapolate
    Transient=BE
    KMC (
        MinLocation = (0.000, 0.000, 0.000)
        MaxLocation = (0.100, 0.100, 0.100)
        CellLength = 0.0005
        Trap2ElecMinDist = 0.0003
        Trap2TrapMinDist = 0.0003
        RandomSeed=1

        ResetElectronFill
        UseAllCrossings
        #MoreFrequentRateResets

        IntegrationPoints = 128
        SkipKMCCurrentCheck = 0.10
    )
}
Solve {
    Coupled {Poisson}

    Quasistationary (
        InitialStep=0.1 MinStep=1e-3 MaxStep=0.5
        Goal {Name="anode" Voltage=4.2}
```

## Chapter 36: Kinetic Monte Carlo MIM Transport

Example: TiN/ZrO<sub>2</sub>/TiN KMC-TDBB Simulation

```
) { Coupled {Poisson} }

NewCurrentFile="tddb_zro2_4.2_"
Transient (
    InitialTime=0 FinalTime=2000 Increment=1.2 InitialStep=1e-12
    MaxStep=1e30
) { Coupled {Poisson Contact}
    Plot ( FilePrefix="tddb_zro2_4.2" Time=(range = (1 1000))
           NoOverwrite )
}

NewCurrentFile="tmp_"
Quasistationary (
    InitialStep=0.1 MinStep=1e-3 MaxStep=0.5
    Goal {Name="anode" Voltage=4.4}
) { Coupled {Poisson} }

NewCurrentFile="tddb_zro2_4.4_"
Transient (
    InitialTime=0 FinalTime=200 Increment=1.2 InitialStep=1e-12
    MaxStep=1e30
) { Coupled {Poisson Contact} }

NewCurrentFile="tmp_"
Quasistationary (
    InitialStep=0.1 MinStep=1e-3 MaxStep=0.5
    Goal {Name="anode" Voltage=4.6}
) { Coupled {Poisson} }

NewCurrentFile="tddb_zro2_4.6_"
Transient (
    InitialTime=0 FinalTime=20 Increment=1.2 InitialStep=1e-12
    MaxStep=1e30
) { Coupled {Poisson Contact} }
}
```

---

## Sentaurus Device Parameter File

```
# tddb.par
Material = "TiN" {
    Bandgap {
        WorkFunction = 4.4      # [eV]
        FermiEnergy  = 6.0      # [eV]
    }
    KMC_MIM_Transport {
        metun = 1.00            # [m0]
        medos = 1.00            # [m0]
    }
}
Material = "ZrO2" {
    KMC_MIM_Transport {
```

## Chapter 36: Kinetic Monte Carlo MIM Transport

Example: TiN/ZrO<sub>2</sub>/TiN KMC-TDBB Simulation

```
metun      = 0.5          # [m0]
medos      = 0.5          # [m0]
mcb        = 1.16         # [m0]
mvb        = 2.5          # [m0]
HuangRhys = 23           # [1]
PhononEnergy = 0.040     # [eV]
OpticalPermittivity = 5.6 # [1]
PhononFrequency = 1e13    # [s^-1]
alpha       = 1             # [1]
tau0       = 1.e-10        # [s]
nu_max     = 1.5e13        # [s^-1]
EA          = 3.0           # [eV]
p0          = 2.05          # [eA]
SitesPerVolume = 5.55e22   # [cm^-3]
}
Epsilon {
    epsilon = 40.0          # [1]
}
Bandgap {
    Chi0     = 2.5           # [eV] CBO = 4.4 - 2.5 = 1.9
    Eg0     = 5.4             # [eV]
    Alpha   = 0.0            # [eV K^-1], no temperature dependence
}
}
```

---

## Results

Figure 82 shows the MIM current as a function of time for anode biases of 4.2 V, 4.4 V, and 4.6 V. Initially, each curve shows the leakage current obtained from the initial configuration of defects. The leakage current increases as defects are created as a function of time.

Although you cannot observe a hard breakdown with the present simulation capabilities, you can assume that, after a certain current level is reached, the device will rapidly go into a hard breakdown state. If you choose a time when the current increases by a factor of 10 from the initial leakage current as that level in this simulation, then you will find that the time to breakdown is approximately 630 s, 80 s, and 8 s for V(node) = 4.2 V, 4.4 V, and 4.6 V, respectively. This is qualitatively similar to the breakdown times shown in [6].

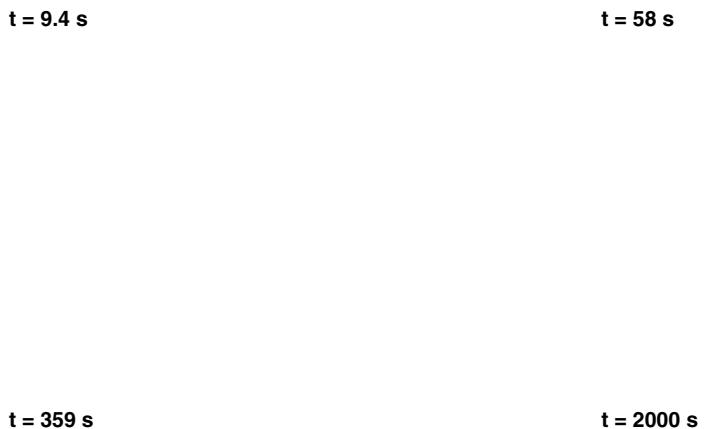
## Chapter 36: Kinetic Monte Carlo MIM Transport

Example: TiN/ZrO<sub>2</sub>/TiN KMC-TDDB Simulation

Figure 82 MIM current versus time for  $V(\text{anode}) = 4.2 \text{ V}, 4.4 \text{ V}, \text{ and } 4.6 \text{ V}$

[Figure 83](#) shows the defects in the MIM structure for  $V(\text{node}) = 4.2 \text{ V}$  at four different times during the generation process.

**Figure 83** Defects in the MIM structure with  $V(\text{anode}) = 4.2 \text{ V}$  at times (upper left) 9.4 s, (upper right) 58 s, (lower left) 359 s, and (lower right) 2000 s



---

## References

- [1] G. C. Jegert, *Modeling of Leakage Currents in High-k Dielectrics*, PhD thesis, Technische Universität München, Institut für Nanoelektronik, Munich, Germany, September 2011.
- [2] M. Herrmann and A. Schenk, "Field and High-temperature Dependence of the Long Term Charge Loss in Erasable Programmable Read Only Memories: Measurements and Modeling," *Journal of Applied Physics*, vol. 77, no. 9, pp. 4522–4540, 1995.
- [3] L. Larcher, "Statistical Simulation of Leakage Currents in MOS and Flash Memory Devices With a New Multiphonon Trap-Assisted Tunneling Model," *IEEE Transactions on Electron Devices*, vol. 50, no. 5, pp. 1246–1253, 2003.
- [4] A. Padovani and L. Larcher, "Time-Dependent Dielectric Breakdown Statistics in  $\text{SiO}_2$  and  $\text{HfO}_2$  Dielectrics: Insights from a Multi-Scale Modeling Approach," in *International Reliability Physics Symposium (IRPS)*, Burlingame, CA, USA, pp. 3A.2-1–3A.2-7, March 2018.

## Chapter 36: Kinetic Monte Carlo MIM Transport

### References

- [5] H.-M. Kwon *et al.*, “A Correlation Between Oxygen Vacancies and Reliability Characteristics in a Single Zirconium Oxide Metal–Insulator–Metal Capacitor,” *IEEE Transactions on Electron Devices*, vol. 61, no. 8, pp. 2619–2627, 2014.
- [6] W. Y. Choi *et al.*, “A Technology-Computer-Aided-Design-Based Reliability Prediction Model for DRAM Storage Capacitors,” *Micromachines*, vol. 10, no. 4, p. 256, 2019.
- [7] A. Padovani *et al.*, “A microscopic mechanism of dielectric breakdown in SiO<sub>2</sub> films: An insight from multi-scale modeling,” *Journal of Applied Physics*, vol. 121, no. 15, p. 155101, 2017.
- [8] E. Y. Wu, “Facts and Myths of Dielectric Breakdown Processes—Part I: Statistics, Experimental, and Physical Acceleration Models,” *IEEE Transactions on Electron Devices*, vol. 66, no. 11, pp. 4523–4534, 2019.
- [9] R. Degraeve *et al.*, “New Insights in the Relation Between Electron Trap Generation and the Statistical Properties of Oxide Breakdown,” *IEEE Transactions on Electron Devices*, vol. 45, no. 4, pp. 904–911, 1998.

# 37

## Kinetic Monte Carlo ReRAM

---

This chapter describes how to study resistive random access memory (ReRAM) structures at the microscopic level by providing an interface to the kinetic Monte Carlo (KMC) engine.

---

### Starting the Simulation and Specifying Particles

ReRAM simulations are launched by specifying `KmcReram` in the `Traps` statement of the `Physics` section. For example:

```
Physics{
    Traps(( KmcReram ))
}
```

Defects are included in the KMC-ReRAM simulation through specifications in the `KMCDefects` section of the Sentaurus Device command file. The defects can be defined as conductive particles (with keyword `Filament`) or nonconductive particles (with keyword `Particle`). Sentaurus Device supports three different types of particle and filament.

The particle or filament can have basic properties such as names and relations with other particles or filaments. The relation among particles can be one of the following:

- The default *non-interactive relation* suggests that different types of defect can be located at the same location.
- The *exclusive relation* (defined by the keyword `Exclude`) suggests different types of trap cannot be located at the same location.
- The *shared capacity relation* (defined by the keyword `ShareCapacity`) suggests that different types of trap can be located at the same location but the total number cannot be greater than the capacity.

The following example defines an oxygen, a vacancy with charge, and a vacancy without charge. The vacancies without charge are conductive particles and cannot share the same location with other types of particle:

```
Physics{
    KMCDefects(
        Particle1( # Define particle 1
```

## Chapter 37: Kinetic Monte Carlo ReRAM

### Starting the Simulation and Specifying Particles

```

        Name="Oxygen"          # Three particle types can be defined
        Exclude(Filament1)
    )
Particle2(
    Name="Vacancy2+"
    Exclude(Filament1)
)

Filament1(
    Name="Vacancy0"        # Define filament 1
    # Three filament particle types can be
    # defined
    Exclude(Particle1 Particle2)
)
)
}

```

where:

- The `KMCDefects/Particle{i}` or `KMCReRAM/Filament{i}` statement defines the particle type. The particles defined by the keywords `Filament1`, `Filament2`, and `Filament3` are conductive particles and are considered in the conductance equation.
- In the `KMCDefects/Particle{i}` statements:
  - `Name` specifies the name of the particle. Default: empty string.
  - `Exclude` specifies a list of particles to be excluded. The list can include `Particle2` or `Filament2`, or `Particle3` or `Filament3`, or both. The defects exclude each other and, therefore, they cannot be located in the same KMC cell.
  - `ShareCapacity` specifies a list of particles with a shared capacity relation. The list can include `Particle2`, or `Particle3`, or both.

## Capacity of Defects

In three dimensions, the capacity of particles is defined by `ParticlesPerVolume{i}` or `FilamentsPerVolume{i}` as follows:

$$\text{capacity} = \text{ParticlesPerVolume} \times \text{CellLength}^3 \frac{10^{-4} \text{ cm}^3}{\mu\text{m}^3} \quad (1260)$$

where `SitesPerVolume` is an insulator property that is specified in the parameter file (default = 4.40e22 (SiO2), 5.54e22 (HfO2), and 5.55e22 (ZrO2) in units of  $\text{cm}^{-3}$ ).

## KMC Events

Based on the defects defined, you can define KMC events (processes) during ReRAM operations. [Table 186](#) lists the supported physical events.

*Table 186     Supported physical events*

Physical event	Description	Relevant equation
Diffusion	<b>Keywords:</b> Diffusion A particle moves to a neighboring cell.	$A(r_1) \rightarrow A(r_2)$
Bulk generation or recombination	<b>Keywords:</b> Generation, Recombination One or two particle types are generated at a location or neighboring locations.	$0 \leftrightarrow A(r) + B(r)$
Interface generation or recombination	<b>Keywords:</b> Generation, Recombination One or two particle types are generated or recombined at the interface.  For generation or recombination of two particles, the particles are located on different sides of the interface.  For generation or recombination of one particle, the particle is located on one side of the interface.	$0 \leftrightarrow A(r_1) + B(r_2)$
Filament growth or recession	<b>Keywords:</b> FilamentGrowth, FilamentRecession Transition between particle and filament.	$F(r) \leftrightarrow A(r)$

For the event rate, a generic expression is used:

$$r = v \exp\left[-\frac{E_A - pF}{k_B T}\right] \quad (1261)$$

where:

- $v$  is the maximum rate of diffusion (keyword `Frequency`; default is 0.0).
- $E_A$  is the activation energy (keyword `Ea`; default is 0.0).
- $p$  is the molecular dipole (keyword `Dipole`; default is 0.0).
- $F$  is the electric field.

The  $v$ ,  $E_A$ , and  $p$  are parameters to be determined.

**Note:**

By default, the KMC event rate is 0.

---

## Location of KMC Events

The location of KMC events is specified when the `Physics` section is created. For example, `Physics(MaterialInterface="Ti/HfO2"){}{}` can be used to create interface generation and recombination, and `Physics(Material="HfO2"){}{}` can be used to create all events in the region of HfO<sub>2</sub>.

---

## Particle or Particle Pairs Involved in KMC Events

The particle or particle pairs must be specified as an option in the `Physics` section. For particles in diffusion, filament growth, or filament recession events, the particles or filaments defined in the particle specification section are sufficient.

For particles involved in generation or recombination events, the `FrenkelPair` option must be defined. Particles or filaments forming the Frenkel pair and regions where the defects are located should also be specified as follows:

- `FrenkelPair1` is the Frenkel pair involved in generation or recombination. The Frenkel pair can be defined on interfaces (two regions) or in the same region. Sentaurus Device supports a maximum of three different Frenkel pairs.
- `FrenkelPair1 or Particle1` is the particle in the Frenkel pair.
- In the `FrenkelPair1 or Particle1` statement, `Region` is the region of the particle in the Frenkel pair. This applies to interface Frenkel pairs.
- In the `FrenkelPair1` statement, `SameSite` specifies that two particles are at the same location (used for bulk generation or recombination only).

The following example defines a Frenkel pair at the Ti–HfO<sub>2</sub> interface:

```
FrenkelPair1(  
    Particle1(Region="Titanium")  
    Particle2(Region="HfO2")  
)
```

## Parameters of KMC Events

For every KMC event, you specify the keywords `Frequency`, `Ea`, and `Dipole` in the command file. For example:

```
Physics (Material = "Titanium") {  
    KMCDefects (  
        Diffusion(Particle1(Frequency=1e10 Ea=0.4 Dipole=1e-8))  
    )  
}
```

## Diffusion Events

Diffusion events can be defined for any particle within a material or a region. The following example is the specification of a diffusion event for `Particle1`:

```
Diffusion(Particle1(Frequency=1e10 Ea=0.4 Dipole=1e-8))
```

Here, `Diffusion` defines a diffusion event for `Particle{i}`, which can be `Particle1`, `Particle2`, and `Particle3`.

## Generation and Recombination Events

Generation and recombination events can be defined for a particle pair within a region or an interface. The Frenkel pair is a pair of two particles.

For bulk generation or recombination, the Frenkel pair should be defined at the same region; for interface generation or recombination, the Frenkel pair should be defined in different regions.

For example:

```
FrenkelPair1(Particle1(Region="Titanium") Particle2(Region="HfO2"))  
  
Generation(FrenkelPair1(Frequency=1e10 Ea=0.7 Dipole=1e-7))  
  
Recombination(FrenkelPair1(Frequency=1e10 Ea=0.3 Dipole=1e-7))
```

where:

- `FrenkelPair1(...)` defines the Frenkel pair involved in the generation or recombination event. The Frenkel pair can be defined on interfaces (two regions) or in the same region.
- `FrenkelPair1(Particle1 ...)` defines the particle in the Frenkel pair.
- In the `FrenkelPair1(Particle1 ...)` statement, `Region` specifies the region of the particle in the Frenkel pair. This applies to interface Frenkel pairs.
- `Generation` indicates a generation event is activated.

- Recombination indicates a recombination event is activated.
- Generation(FrenkelPair1 ...) defines a generation event for FrenkelPair1.
- Recombination(FrenkelPair1 ...) defines a recombination event for FrenkelPair1.

## Filament Growth and Recession Events

Filament growth and recession events can be defined for any filament, and the particle type transformed to a filament must be specified.

For example:

```
FilamentGrowth(Filament1(Frequency=1e9 Ea=0.6 Dipole=1e-7 Particle2))  
  
FilamentRecession(Filament1(Frequency=1e9 Ea=0.55 Dipole=1e-7  
Particle2))
```

where:

- FilamentGrowth indicates that a growth event is activated.
- FilamentRecession indicates that a recession event is activated.
- FilamentGrowth(Filament1 ...) defines the growth of Filament1.
- FilamentRecession(Filament1 ...) defines the recession of Filament1.

## Seeds for Filament Growth

Filament growth can commence from preexisting filament particles or electrodes. You can use the FilamentSeeds statement to specify the starting point of filament growth.

For example:

```
Physics (Electrode = "anode") {  
    KMCDefects(  
        FilamentSeeds(Filament1)    # Define anode as seed for Filament1  
    )  
}
```

where FilamentSeeds(Filament1) defines the seed for Filament1. Filament growth commences at that point.

---

## KMC Simulation Space and Math Settings

A typical ReRAM calculation requires a `KMC` statement in the `Math` section. Options for the KMC space, random seed, time step control, and current compliance must be included. For example:

```
Math {
    KMC (
        MinLocation = (0.00, 0.00, 0.00)      # Minimum location of KMC box
        MaxLocation = (0.02, 0.02, 0.02)      # Maximum location of KMC box
        CellLength = 0.0005                   # KMC grid size
        RandomSeed = 1                      # Random seed
        TargetParticle(Filament1)           # Time step is determined by
                                            # tracking Filament1
        UseKMCTimeStep                  # KMC determines the time step
        MaxTrapNumberChange = 1            # In each KMC time step, 'TargetParticle'
                                            # number changes by 1 at most
        MaxTrapNumber = 100               # Stop transient calculation whenever
                                            # 'TargetParticle' reaches the
                                            # maximum trap number
        CurrentCompliance(
            Contact = "anode"          # Use current compliance
            minval = -1e-7              # One contact must be specified
            maxval = 1e-7
        )
    )
}
```

---

## KMC Simulation Space

The portion of the Sentaurus Device structure that is used for the KMC-ReRAM simulations must be defined with the following keywords in the `Math` section:

- `MinLocation` specifies the minimum location for the KMC calculation.
- `MaxLocation` specifies the maximum location for the KMC calculation.
- `CellLength` sets the grid size of the KMC region defined by `MinLocation` and `MaxLocation`.

These keywords define the extremities of a KMC box within the Sentaurus Device structure. If part of the specification lies outside of the Sentaurus Device structure, then it is ignored.

Other details of the KMC simulation space are:

- It must include at least one insulator region.
- It must include at least two electrodes.

## Chapter 37: Kinetic Monte Carlo ReRAM

### KMC Simulation Space and Math Settings

- If metal regions are used for contacting the insulator regions, then an electrode must contact each metal region. In addition, both the Poisson and contact equations must be included when obtaining solutions.
- It supports 2D and 3D structures.

The following example specifies a 10 nm × 10 nm × 10 nm KMC simulation space:

```
Math {
    KMC (
        MinLocation = (0.000, 0.000, 0.000)
        MaxLocation = (0.010, 0.010, 0.010)
    )
}
```

---

## Random Seed

KMC simulations are stochastic and each execution of a command file typically produces results that differ to some degree. Specifying a fixed `RandomSeed` greater than zero allows the same results to be retrieved from one simulation to the next, assuming the command file and parameters have not changed and that the simulation is performed on the same platform.

---

## KMC Time Step Control

KMC event rates depend on the electric field and temperature. Therefore, whenever particles change, fields and temperatures must be recalculated.

To handle abrupt changes during the forming, setup, and reset process, the option `UseKMCTimeStep` can be used to instruct the KMC simulator to update fields with the correct time step. With this option, Sentaurus Device determines the time step based on stochastic KMC time steps rather than the time step specified in the `Transient` section.

Some types of particle have little impact on electric fields because they are in metal regions, and other particles have a greater impact on electric fields. The `TargetParticle` statement can be used to select the particles that the KMC simulator is monitoring; the KMC time step is determined based on changes to `TargetParticle`. The option `MaxTrapNumberChange` specifies the maximum `TargetParticle` number of changes within a single KMC time step.

In summary:

- `UseKMCTimeStep` instructs the KMC simulator to determine the time step in transient calculations.
- `TargetParticle/Filament{i}` defines the particles used to determine the time step.

- `MaxTrapNumberChange` specifies the maximum trap number changes for a transient time step. Default: not applied.
- `MaxTrapNumber` specifies the maximum number of traps used to terminate the KMC calculation. Default: not applied.

---

## Current Compliance

ReRAM is usually operated with a current compliance. The current compliance limits the maximum current during the forming and setup process. With a current compliance, the size of the filament can be controlled.

You can use the option `CurrentCompliance` to limit the maximum current of a selected electrode as follows:

- `CurrentCompliance` specifies to use current compliance.
- `CurrentCompliance(Contact=<string>)` specifies the name of the contact applying current compliance. Default: empty string.
- `CurrentCompliance(MinVal=<float>)` specifies the minimum value accepted. The value must be negative. Default: not applied.
- `CurrentCompliance(MaxVal=<float>)` specifies the maximum value accepted. The value must be positive. Default: not applied.

When the current compliance is reached, the biases applied to the ReRAM change so that the current is maintained at the `MaxVal` or `MinVal` specified.

**Note:**

The KMC process continues even if the current reaches `MaxVal` or `MinVal`. The size of the filament and the defect distribution will continue to change.

For example:

```
Math {
    KMC (
        CurrentCompliance(      # Use current compliance.
            Contact = "anode"   # Only one contact needs to be specified.
            MinVal = -1e-7
            MaxVal = 1e-7
        )
    )
}
```

---

## Specifying the Material

Insulator properties for a KMC-ReRAM simulation are specified in the parameter file. The parameters related to the capacity, heat equation, and conductance equation are defined here. For example:

```
Material = "HfO2" {
    FilamentParameter {
        SigmaDef1 = 0.001          # [S/cm]
        Sigma = 1e-8               # [S/cm]
        KappaDef1 = 0.1             # [W/cm/K]
        Kappa = 0.1                # [W/cm/K]
        ParticlesPerVolume1 = 4e22  # [cm^-3]
        FilamentsPerVolume1 = 4e22 # [cm^-3]
    }
}
```

where:

- `SigmaDef{i}` sets the conductivity of the particle of `Filament{i}`. Default: 10 S/cm
- `Sigma` sets the conductivity of the material. Default: 1e-11 S/cm
- `KappaDef{i}` sets the thermal conductivity of the particle of `Filament{i}`. Default: 0.23 W/cm/K
- `Kappa` sets the spatially dependent thermal conductivity of the material. Default: 0.11 W/cm/K
- `ParticlesPerVolume{i}` sets the volume of `Particle{i}`. Default: 4.4e22 cm<sup>-3</sup>
- `FilamentsPerVolume{i}` sets the volume of `Filament{i}`. Default: 4.4e22 cm<sup>-3</sup>

**Note:**

`SigmaDef` and `KappaDef` apply only to filament particles.

---

## Specifying the Field and Temperature

When you select the `TDDB` option, the `ConductanceEquation` and `HeatEquation` options are switched on by default. The steady-state conductance equation is given by:

$$\vec{\nabla} \cdot \vec{J} = \vec{\nabla} \cdot (\sigma \vec{\nabla} \psi) = 0 \quad (1262)$$

## Chapter 37: Kinetic Monte Carlo ReRAM

### Transient Calculation for ReRAM

The steady-state heat equation with Joule heating is given by:

$$\vec{\nabla} \cdot \kappa \vec{\nabla} T + \psi \sigma \vec{\nabla} \psi = 0 \quad (1263)$$

In these equations,  $\psi$  is the potential,  $T$  is the temperature, and  $\sigma$  and  $\kappa$  are the spatially dependent electrical and thermal conductivity that account for the presence of defects in KMC cells.

You can use the `UpdateKMCTemperature` option to selectively update the temperature in the KMC rate calculation. In other words, without this option, the KMC rate calculation still uses the initial temperature although the heat equation produces different temperatures.

The solution of these equations over the KMC cells will provide better values for the potential (and electric field) and temperature to use in the generation expression. If you specify `-HeatEquation`, then the heat equation is not solved. If you specify `-ConductanceEquation`, then neither equation is solved.

For example:

```
Physics{
    KMCDefects(
        ConductanceEquation
        HeatEquation
        UpdateKMCTemperature
    )
}
```

where:

- `ConductanceEquation` specifies to solve the conductance equation for the potential.
- `HeatEquation` specifies to solve the heat equation to determine the temperature.
- `UpdateKMCTemperature` specifies to use the temperature obtained from the heat equation in the KMC simulation. Otherwise, use the initial temperature.

---

## Transient Calculation for ReRAM

To sweep the bias of an electrode, in the `Transient` statement, use options in the `Goal` statement. For example:

```
Transient (
    InitialTime=0 FinalTime=0.1 Increment=1.2
    InitialStep=0.001 MaxStep=1e30
    Goal{Name="anode" Voltage=3.0}
) { Coupled { Poisson Contact } }
```

## Current in ReRAM Operations

This section discusses different types of current relevant to ReRAM operations.

---

### Conduction Current

The conduction current is calculated when the conductance equation ([Equation 1262](#)) is solved.

---

### Trap-Assisted Tunneling Calculation

To activate the trap-assisted tunneling calculation, specify:

```
KMCDefects(TrapAssistedTunneling)
```

### Particles for Trap-Assisted Tunneling Calculation

The ReRAM solver is used to obtain the trap-assisted tunneling current and the conduction current.

To link the particle and filament defined for the ReRAM process, use the option `kmcmim1` or `kmcmim2` with other MIM-related options for traps. For example:

```
Particle2(Name="Vacancy" kmcmim1 EtFill=1.1 EtEmpty=1.1  
Xsection=1e-14 XsectionT2T=1e-14)  
  
Filament1(Name="ImmobileVacancy" kmcmim2 EtFill=1.10 EtEmpty=1.1  
Xsection=1e-14 XsectionT2T=1e-14)
```

Here, the keyword for trap-assisted tunneling calculations is repeated (see [Chapter 36 on page 1101](#)), and where:

- `kmcmim1` and `kmcmim2` specify the defect types for the trap-assisted tunneling calculation as defined in the MIM calculation.
- `EtFill` sets the filled trap level measured from the conduction band.
- `EtEmpty` sets the empty trap level measured from the conduction band.
- `TrapSigma`, if specified, means the trap energy levels are randomized using a Gaussian distribution with this standard deviation.
- `Xsection`, if specified, means this value is used to calculate the defect localization radius used in the inelastic electrode ↔ trap rate calculations using  $r_D = \sqrt{Xsection/\pi}$ .

## Chapter 37: Kinetic Monte Carlo ReRAM

### Current in ReRAM Operations

- `XsectionT2T`, if specified, means this value is used to calculate the defect localization radius used in the multiphonon trap-to-trap rate calculations using  
 $r_D = \sqrt{XsectionT2T/\pi}$ .

## Physics for Trap-Assisted Tunneling Calculation

The physics included in the trap-assisted tunneling calculation should be specified in the `KMC_MIM_Transport` statement in the `Physics` section (see [Chapter 36 on page 1101](#)) as follows:

- `DirectTunneling`: Include direct tunneling in the trap-assisted tunneling calculation.
- `PooleFrenkelEmission`: Include Poole–Frenkel emission in the trap-assisted tunneling calculation.
- `DeltaTrapPotential`: Assume trap potential as a delta potential. This option affects the prefactor of trap-to-trap, electrode-to-trap, and trap-to-electrode tunneling.
- `TrapToTrapTunneling [(InelasticPhonon) | (MultiPhonon)]`: Include trap-to-trap tunneling. You can use the elastic (default), inelastic, or multiphonon model.
- `ElasticElectrodeToTrapTunneling`: Include the elastic electrode-to-trap tunneling model.
- `ElasticTrapToElectrodeTunneling`: Include the elastic trap-to-electrode tunneling model.
- `InelasticElectrodeToTrapTunneling`: Include the inelastic (multiphonon) electrode-to-trap tunneling model.
- `InelasticTrapToElectrodeTunneling`: Include the inelastic (multiphonon) trap-to-electrode tunneling model.
- `InelasticIncludePZero`: Ignore the  $p < 0$  terms in the inelastic model.
- `InelasticIgnorePMinus`: Include the  $p = 0$  term in the inelastic model.
- `InelasticPZeroOnly`: Use only the  $p = 0$  term in the inelastic model.
- `ImageChargeBarrierLowering`: Consider image-charge barrier-lowering effects.
- `EnergyDependenceMass`: Use the energy-dependent mass in all tunneling models.

## Stopping Criteria of Trap-Assisted Tunneling Calculation

Trap-assisted tunneling calculations can be time-consuming when many defects exist. You can use the keyword `MaxTATTrapNumber` to terminate the trap-assisted tunneling calculation when the number of target particles reaches the value specified.

## Chapter 37: Kinetic Monte Carlo ReRAM

### Current in ReRAM Operations

A more effective criterion to terminate the trap-assisted tunneling calculation is the conductance of the device, which can be specified with the keyword `MaxConductance`. The calculation is omitted when the total conductance reaches the value specified.

By default, the trap-assisted tunneling calculation is omitted when current compliance is reached. `MaxTATTTrapNumber` can also be used to stop the trap-assisted tunneling calculation. For example:

```
Math {
    KMC (
        CurrentCompliance (
            Contact = "anode"
            MaxVal = 1E-6
            MaxTATTTrapNumber = 200
            MaxConductance = 1E-8
        )
    )
}
```

## Parameters for Trap-Assisted Tunneling Calculation

Parameters for MIM-TDDB calculations must be included. For example:

```
Material = "TiN" {
    Bandgap {
        Workfunction = 4.4      # [eV] Workfunction
        FermiEnergy = 11.7      # [eV] Fermi energy
    }
    KMC_MIM_Transport {
        metun = 1.0             # [m0] Effective tunneling mass
        medos = 1.0             # [m0] DOS mass
    }
}

# Associate properties of ZrO2 with "Oxide"
Material = "Oxide" {
    Epsilon {
        epsilon = 40.0          # [1] Permittivity
    }
    Bandgap {
        Chi0 = 2.50            # [eV] Electron affinity; CBO = 4.4-2.5 = 1.9
        Eg0  = 5.4              # [eV] Band gap
    }
    KMC_MIM_Transport {
        metun      = 0.50      # [m0] Effective tunneling mass
        medos     = 0.50      # [m0] DOS mass
        mcb       = 1.16      # [m0] Conduction band mass
        mvb       = 2.50      # [m0] Valence band mass
        HuangRhys = 23        # [1] Huang-Rhys parameter
        PhononEnergy = 0.040   # [eV] Phonon energy
        PhononFrequency = 1e13  # [s^-1] Phonon frequency
        OpticalPermittivity = 5.6 # [1] Optical permittivity
    }
}
```

## Chapter 37: Kinetic Monte Carlo ReRAM

### Simulation Output

```
alpha          = 1.0      # [1] InelasticPhonon parameter
tau0         = 1.e-10    # [s] InelasticPhonon fitting parameter
SigmaDef     = 10.0     # [S/cm] Conductivity of cells with defects
SigmaIns     = 1e-11    # [S/cm] Conductivity of empty cells
KappaDef     = 0.23    # [W/cm/K] Thermal conductivity of filled cells
KappaIns     = 0.011   # [W/cm/K] Thermal conductivity of empty cells
}
}
```

---

## Simulation Output

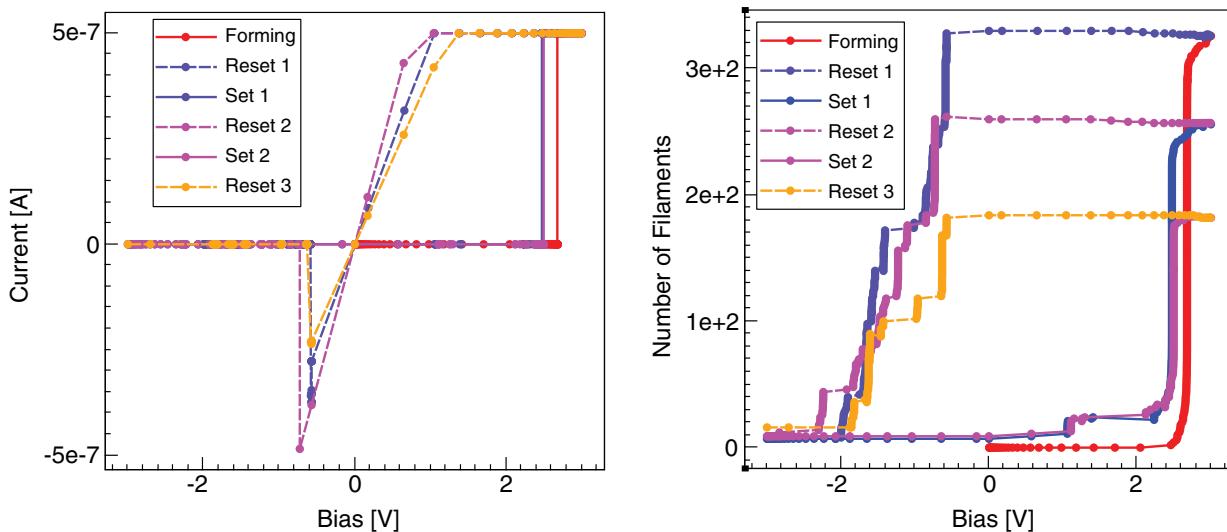
This section discusses the simulation output.

---

### Current and Particles

Plots of the current versus bias, or the number of filaments versus bias, are saved in the .plt file. [Figure 84](#) shows some example plots. For the input files, see [ReRAM Example on page 1169](#).

**Figure 84** Plot of (left) I–V and (right) number of filament particles versus bias during forming, setup, and reset process of a ReRAM operation



---

### Event and Particle Statistics

Specify `PrintMIMEEventStatistics` in the `KMC_MIM_Transport` statement of the `Physics` section to print MIM event statistics to standard output after each Sentaurus Device time step.

For example:

```
KMC Events Summary:  
FrenkelPair1 Bulk Generation count 644  
FrenkelPair1 Bulk Recombination count 122  
FrenkelPair1 Interface recombination count 29  
FrenkelPair2 Interface recombination count 6  
Oxygen diffusion count 17147  
Vacancy diffusion count 8562  
ImmobileVacancy Growth count 27  
ImmobileVacancy Recession count 21  
ImmobileVacancy Growth count 623  
ImmobileVacancy Recession count 250  
  
KMC Particles Summary:  
Number of Oxygen = 567.  
Number of Vacancy = 114.  
Number of ImmobileVacancy = 373.
```

---

## Defect and Field Visualization

You can set up the generation of particle plots of defects in the `Plot` section of the command file by using the `ReRAMParticle` option. The particle plots show the actual position of the defects within the Sentaurus Device structure when visualized with Sentaurus Visual.

When viewing the TDR file in Sentaurus Visual:

- The particles to be displayed can be chosen on the **Lines/Particles** tab of the Selection panel.
- The size and color of particles can be modified by choosing **Data > Region Properties**.

For example:

```
Plot {  
    ...  
    ReRAMParticle  
    ReRAMPotential  
    ReRAMTemperature  
    ReRAMField  
    ReRAMCurrent  
}
```

where:

- `ReRAMParticle` specifies to output a 3D discrete particle view of the ReRAM.
- `ReRAMPotential` specifies to output the potential in the 3D structure; it appears in the TDR file.

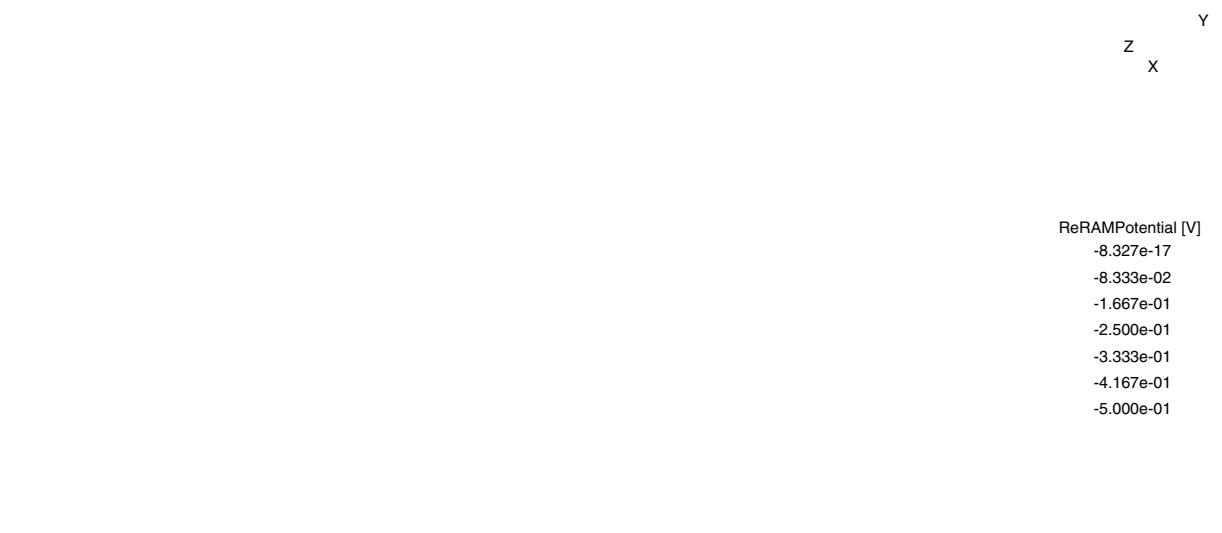
## Chapter 37: Kinetic Monte Carlo ReRAM

### ReRAM Example

- `ReRAMTemperature` specifies to output the temperature in the 3D structure; it appears in the TDR file.
- `ReRAMField` specifies to output the electric field in the 3D structure; it appears in the TDR file.
- `ReRAMCurrent` specifies to output the conductance current in the 3D structure; it appears in the TDR file.

[Figure 85](#) shows examples of visualization of output. For the input files, see [ReRAM Example](#).

Figure 85 (Left) Particle distribution in ReRAM and (right) potential distribution in ReRAM



---

## ReRAM Example

For this example, the structure of the ReRAM is shown in [Figure 85 \(left\)](#). The ReRAM contains four layers: the top layer is  $\text{HfO}_2$ , the bottom layer is titanium, and the two thin layers in the middle are  $\text{HfOx}$  and  $\text{TiOx}$ . The two thin layers approximate an imperfect  $\text{HfO}_2$ – $\text{Ti}$  interface. In this example, trap generation and recombination occur in the  $\text{HfOx}$  layer.

---

## Sentaurus Device Command File

```
# mim_3d.cmd
File {
    Grid      = "reram_structure_msh.tdr"
    Plot      = "reram_3d_thin"
    Current   = "reram_3d_thin"
```

## Chapter 37: Kinetic Monte Carlo ReRAM

### ReRAM Example

```
Output    = "reram_3d_thin"
Param     = "reram_3d.par"
}

Plot {
    VertexIndex
    ReRAMParticle
    ReRAMPotential
    ReRAMTemperature
    ReRAMField
    ReRAMCurrent
}

Electrode {
    {name="cathode" voltage=0.0 Material="TiN"}
    {name="anode"   voltage=0.0 Material="TiN"}
}

Thermode {
    {name="cathode" temperature=298}
    {name="anode"   temperature=298}
}

Physics {
    Temperature=298
    Traps(( KmcReram ))
    KMCDefects(
        ConductanceEquation
        HeatEquation
        Particle1(Name="Oxygen")
        Particle2(Name="Vacancy")
        Filament1(Name="ImmobileVacancy")
    )
}

Physics (Material = "HfO2") {
    KMCDefects(
        Diffusion(Particle2(Frequency=1e10 Ea=0.4 Dipole=2e-7))
        FilamentGrowth(Filament1(Frequency=1e9 Ea=0.4 Dipole=2e-7
            Particle2))
        FilamentRecession(Filament1(Frequency=1e9 Ea=0.5 Dipole=2e-7
            Particle2))
    )
}

Physics (Material="HfOx") {
    KMCDefects(
        FrenkelPair1(Particle1(Region="Oxide") Particle2(Region="Oxide")
            SameSite)
        Generation(FrenkelPair1(Frequency=1e8 Ea=0.8 Dipole=2e-7))
        Recombination(FrenkelPair1(Frequency=1e9 Ea=0.3 Dipole=0e-7))
        Diffusion(Particle1(Frequency=1e10 Ea=0.4 Dipole=-2e-7))
        Diffusion(Particle2(Frequency=1e10 Ea=0.4 Dipole=2e-7))
    )
}
```

## Chapter 37: Kinetic Monte Carlo ReRAM

### ReRAM Example

```
    FilamentGrowth(Filament1(Frequency=1e9 Ea=0.4 Dipole=2e-7
                                Particle2))
    FilamentRecession(Filament1(Frequency=1e9 Ea=0.5 Dipole=2e-7
                                Particle2))
)
}

Physics (MaterialInterface="TiOx/HfOx") {
    KMCDefects(
        FrenkelPair1(Particle1(Region="Metal"
                               Particle2(Region="Oxide")))
        Recombination(FrenkelPair1(Frequency=1e9 Ea=0.4 Dipole=4e-7))
        FrenkelPair2(Particle1(Region="Metal"
                               Filament1(Region="Oxide")))
        Recombination(FrenkelPair2(Frequency=1e9 Ea=0.4 Dipole=4e-7))
    )
}

Physics (Material="TiOx") {
    KMCDefects(
        Diffusion(Particle1(Frequency=1e10 Ea=0.4 Dipole=-2e-7))
    )
}

Physics (Material = "Titanium") {
    KMCDefects(
        # Particle1(Conc=1e21)
        Diffusion(Particle1(Frequency=1e8 Ea=0.4 Dipole=-2e-7))
    )
}

Physics (Electrode = "cathode") {
    KMCDefects(
        FilamentSeeds(Filament1)
    )
}

Physics (Electrode = "anode") {
    KMCDefects(
        FilamentSeeds(Filament1)
    )
}

Math {
    Extrapolate
    NumberOfThreads=1
    Iterations=200
    transient=BE
    KMC (
        CurrentCompliance (
            Contact = "anode" maxval = 1E-8
        )
        MinLocation = (0.00, 0.00, 0.00)
```

## Chapter 37: Kinetic Monte Carlo ReRAM

### ReRAM Example

```
    MaxLocation = (0.02, 0.02, 0.02)
    CellLength = 0.0005
    UseKMCTimeStep
    MaxTrapNumberChange = 1
    MaxTrapNumber = 1000
    TargetParticle(Filament1)
)
}

Solve {

    Coupled (iterations=100) {Poisson Contact}

    NewCurrentFile="tmp"
    Quasistationary (
        InitialStep=0.1 MinStep=1e-3 MaxStep=0.5
        Goal {Name="anode" Voltage=0.0}
    ) { Coupled { Poisson Contact} }

    NewCurrentFile="V-1"
    Transient (
        InitialTime=0 FinalTime=0.015 Increment=1.2 InitialStep=0.001
        MaxStep=1e30

        Goal{Name="anode" Voltage=3.0}
    ) { Coupled { Poisson Contact }
        Plot(FilePrefix="reram_thin_1" Time=(range = (1e-10 0.01))
            NoOverwrite)
    }

    NewCurrentFile="tmp"
    Quasistationary (
        InitialStep=0.01 MinStep=1e-3 MaxStep=0.5
        Goal {Name="anode" Voltage=3.0}
    ) { Coupled { Poisson Contact} }
    NewCurrentFile="V-2"
    Transient (
        InitialTime=0.0 FinalTime=0.03 Increment=1.2 InitialStep=0.001
        MaxStep=1e30
        Goal{Name="anode" Voltage=-3.0}
    ) { Coupled { Poisson Contact }
        Plot(FilePrefix="reram_thin_2" Time=(range = (1e-7 0.1))
            NoOverwrite)
    }

    NewCurrentFile="tmp"
    Quasistationary (
        InitialStep=0.01 MinStep=1e-3 MaxStep=0.5
        Goal {Name="anode" Voltage=-3.0}
    ) { Coupled { Poisson Contact} }
    NewCurrentFile="V-3"
```

## Chapter 37: Kinetic Monte Carlo ReRAM

### ReRAM Example

```
Transient (
    InitialTime=0.0 FinalTime=0.03 Increment=1.2 InitialStep=0.001
    MaxStep=1e30
    Goal{ Name="anode" Voltage=3.0 }
)
{ Coupled { Poisson Contact }
    Plot(FilePrefix="reram_thin_3" Time=(range = (1e-7 0.1))
        NoOverwrite)
}
}
```

---

## Sentaurus Device Parameter File

```
Material = "TiN" {
    Bandgap {
        WorkFunction = 4.4      # [eV]
        FermiEnergy = 6.0       # [eV]
    }
    FilamentParameter {
        SigmaDef1 = 1e4
        Sigma = 1e4
        KappaDef1 = 0.2
        Kappa = 0.2
    }
}

Material = "Titanium" {
    Bandgap {
        WorkFunction = 4.4      # [eV]
        FermiEnergy = 6.0       # [eV]
    }
    FilamentParameter {
        SigmaDef1 = 1e3
        Sigma = 1e3
        KappaDef1 = 0.1
        Kappa = 0.2
        ParticlesPerVolume1 = 5e22
    }
}

Material = "HfOx" {
    FilamentParameter {
        SigmaDef1 = 1
        Sigma = 1e-11
        KappaDef1 = 0.1
        Kappa = 0.1
        ParticlesPerVolume1 = 1.5e22
        ParticlesPerVolume2 = 1.5e22
        FilamentsPerVolume1 = 1.5e22
    }
}
```

## Chapter 37: Kinetic Monte Carlo ReRAM

### ReRAM Example

```
KMC_MIM_Transport {
    metun      = 0.2                      # [m0]
    medos      = 0.2                      # [m0]
    mcb        = 1.16                     # [m0]
    mvb        = 2.5                      # [m0]
    HuangRhys = 17.0                     # [1]
    PhononEnergy = 0.070                  # [eV]
    OpticalPermittivity = 5.6 # [1]
    PhononFrequency = 1e13                # [s^-1]
    tau0       = 1e-10                    # [s]
    nu_max     = 1e14                     # [s^-1]
    EA          = 3.00                     # [eV]
    p0          = 5.2                      # [eA]
}

Epsilon {
    epsilon = 21.0                      # [1]
}

Bandgap {
    Chi0      = 2.4                      # [eV]
    Eg0       = 5.8                      # [eV]
    alpha     = 0.0000e+00               # [eV K^-1]
    beta      = 0.0000e+00               # [K]
    alpha2    = 0.0000e+00               # [eV K^-1]
    beta2    = 0.0000e+00               # [K]
    EgMin    = -1.0000e+01              # [eV]
    dEgMin   = 0.0000e+00               # [eV]
    Tpar      = 0.0000e+00               # [K]
}
}

Material = "TiOx" {
    FilamentParameter {
        SigmaDef1 = 0.01
        Sigma     = 0.01
        KappaDef1 = 0.1
        Kappa     = 0.1
        ParticlesPerVolume1 = 5e22
    }
    Bandgap {
        WorkFunction = 4.7    # [eV]
        FermiEnergy  = 11.7  # [eV]
    }
}

Material = "HfO2" {
    FilamentParameter {
        SigmaDef1 = 0.01
        Sigma     = 1e-11
        KappaDef1 = 0.1
        Kappa     = 0.1
        ParticlesPerVolume2 = 1.5e22
}
```

## Chapter 37: Kinetic Monte Carlo ReRAM

### ReRAM Example

```
        FilamentsPerVolume1 = 1.5e22
    }

    KMC_MIM_Transport {
        metun      = 0.2          # [m0]
        medos     = 0.2          # [m0]
        mcb       = 1.16         # [m0]
        mvb       = 2.5          # [m0]
        HuangRhys = 17.0        # [1]
        PhononEnergy = 0.070     # [eV]
        OpticalPermittivity = 5.6 # [1]
        PhononFrequency = 1e13    # [s^-1]
        tau0      = 1e-10        # [s]
        nu_max   = 1e14         # [s^-1]
        EA        = 3.00         # [eV]
        p0        = 5.2          # [eA]
    }

    Epsilon {
        epsilon = 21.0          # [1]
    }

    Bandgap {
        Chi0      = 2.4          # [eV]
        Eg0      = 5.8          # [eV]
        alpha    = 0.0000e+00    # [eV K^-1]
        beta     = 0.0000e+00    # [K]
        alpha2   = 0.0000e+00    # [eV K^-1]
        beta2   = 0.0000e+00    # [K]
        EgMin   = -1.0000e+01   # [eV]
        dEgMin  = 0.0000e+00    # [eV]
        Tpar    = 0.0000e+00    # [K]
    }
}
```

## **Part III:      Numeric Methods and External Interfaces**

---

This part of the *Sentaurus™ Device User Guide* contains the following chapters:

- [Chapter 38, Numeric Methods](#)
- [Chapter 39, Physical Model Interface](#)
- [Chapter 40, Tcl Interfaces](#)
- [Chapter 41, Python Interface](#)

# 38

## Numeric Methods

---

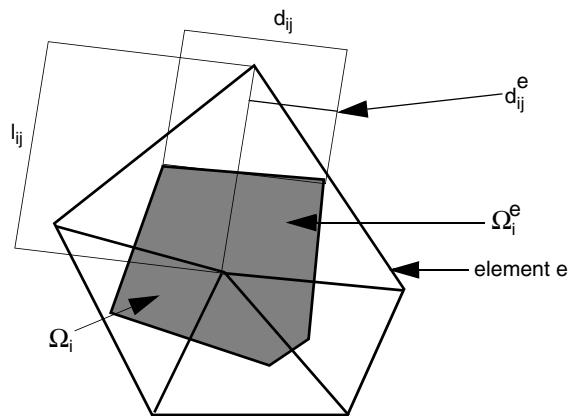
This chapter presents some of the numeric methods used in Sentaurus Device.

---

### Discretization

The well-known *box discretization* [1][2][3] is applied to discretize the partial differential equations (PDEs). This method integrates PDEs over a test volume such as that shown in [Figure 86](#), which applies the Gaussian theorem and discretizes the resulting terms to a first-order approximation.

*Figure 86 Single box for a triangular mesh in two dimensions*



In general, box discretization discretizes each PDE of the form:

$$\nabla \cdot J + R = 0 \quad (1264)$$

into:

$$\sum_{j \neq i} \kappa_{ij} \cdot j_{ij} + \mu(\Omega_i) \cdot r_i = 0 \quad (1265)$$

with values listed in [Table 187 on page 1178](#).

## Chapter 38: Numeric Methods

### Discretization

In this case, the physical parameters  $j_{ij}$  and  $r_i$  have the values listed in [Table 188](#), where  $B(x) = x/(e^x - 1)$  is the Bernoulli function.

*Table 187 Box method parameters: coefficients and control volumes*

Dimension	$\kappa_{ij}$	$\mu(\Omega_i)$
1D	$1/l_{ij}$	Box length
2D	$d_{ij}/l_{ij}$	Box area
3D	$D_{ij}/l_{ij}$	Box volume

*Table 188 Equations*

Equation	$j_{ij}$	$r_i$
Poisson	$\epsilon(u_i - u_j)$	$-\rho_i$
Electron continuity	$\mu_n^n(n_i B(u_i - u_j) - n_j B(u_j - u_i))$	$R_i - G_i + \frac{d}{dt}n_i$
Hole continuity	$\mu_p^p(p_j B(u_j - u_i) - p_i B(u_i - u_j))$	$R_i - G_i + \frac{d}{dt}p_i$
Temperature	$\kappa(T_i - T_j)$	$H_i - \frac{d}{dt}T_i c_i$

One special feature of Sentaurus Device is that the actual assembly of nonlinear equations is performed elementwise, that is:

$$\kappa_{ij}^e \cdot j_{ij}^e + \mu(\Omega_i^e) \cdot r_i^e = 0 \quad (1266)$$

$e \in \text{elements}(i) \quad j \in \text{vertices}(e)$

This expression is equivalent to [Equation 1265](#) but has the advantage that some parameters (such as  $\epsilon$ ,  $\mu_n$ ,  $\mu_p$ ) can be handled elementwise, which is useful for numeric stability and physical exactness. In the 2D case, the box method coefficients have simple visual values:  $\kappa_{ij}^e = d_{ij}^e/l_{ij}$  (see [Figure 86 on page 1177](#)). In the 3D case, these values are not trivial.

## Chapter 38: Numeric Methods

### Box Method Coefficients in the 3D Case

---

## Extended Precision

Although the coordinates of vertices of a mesh are stored with “double” precision accuracy, Sentaurus Device can compute box method coefficients and control volumes with “long double” extended-precision accuracy. This is especially important for the accurate calculation of box method parameters in meshes that contain sliver elements. The procedure is as follows:

1. Read “double” precision coordinates of the mesh vertices.
2. Convert these coordinates to “long double” extended precision.
3. Compute box method parameters with “long double” extended precision.
4. Convert coefficients and control volumes back to “double” precision.

The extended-precision calculation of box method parameters can be switched on by specifying `BM_ExtendedPrecision` in the global `Math` section of the command file.

#### Note:

Extended-precision box method calculations are switched off if the global `Math` section contains either `BoxCoefficientsFromFile`, or `BoxMeasureFromFile`, or `NaturalBoxMethod` (see [Saving and Restoring Box Method Coefficients on page 1187](#)).

---

## Box Method Coefficients in the 3D Case

This section describes the coefficients of the box method in the 3D case.

---

## Basic Definitions

### Delaunay mesh

A mesh is a *Delaunay mesh* if the interior of the circumsphere (circumcircle for two dimensions) of each element contains no mesh vertices.

### Obtuse element

An element is called *obtuse* if the center of the circumsphere (circumcircle) is outside this element.

### Obtuse face

Let  $P_f$  be the plane that contains the face  $f$  of an element. Each plane splits 3D space into two half spaces,  $Sf_1$  and  $Sf_2$ . A face  $f$  is called *obtuse* if the center of the circumsphere of the element and the element itself lie in different half spaces,  $Sf_1$  and  $Sf_2$ .

**Note:**

In the 2D case, an obtuse triangle has only one obtuse edge.

In the 3D case:

- An obtuse prism has only one obtuse face.
- An obtuse tetrahedron has one or two obtuse faces.
- An obtuse pyramid has one, two, or three obtuse faces.

**Non-Delaunay element**

An obtuse element is called *non-Delaunay* if the interior of the circumsphere (circumcircle) around this element contains another mesh vertex.

**Voronoi element center and Voronoi face center**

Let  $T$  be a mesh element. The center circumsphere (circle for two dimensions) around the element  $T$  is called the *Voronoi element center*  $V_T$ . Let  $f$  be the face of the element  $T$ . The center circumcircle around the face  $f$  is called the *Voronoi face center*  $V_f$ .

**Voronoi box and face of the Voronoi box**

Let  $v$  be a vertex of the mesh and let  $ev^n$  ( $1 \leq n \leq N$ ) be the set of edges connected to vertex  $v$ . Let  $P_{ev}^n$  be the mid-perpendicular plane for the edge  $ev^n$ . The plane  $P_{ev}^n$  splits 3D space into two half spaces. Let  $S_{ev}^n$  be the half space that contains the vertex  $v$ . The intersection of all half spaces  $S_{ev}^n$  is called the *Voronoi box*  $B_v$  of vertex  $v$ . Therefore, the Voronoi box  $B_v$  is the convex polyhedron and any face of  $B_v$  is a convex polygon that lies in the mid-perpendicular plane  $P_{ev}^n$ . This face called the *face of the Voronoi box*  $F_{ev}^n$ .

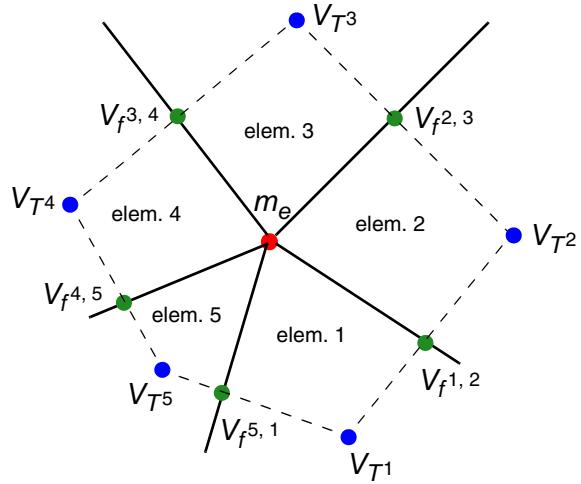
In addition, let  $T_v^m$  ( $1 \leq m \leq M$ ) be the set of elements per vertex  $v$  and let  $T_{ev}^k$  ( $1 \leq k \leq K$ ) be the set of elements per edge  $ev$ . For the Delaunay mesh, the next two propositions hold:

1. Vertices of the Voronoi box  $B_v$  are Voronoi element centers  $(V_{T_v^1}, V_{T_v^2}, \dots, V_{T_v^M})$ .
2. Vertices of the face of the Voronoi box  $F_{ev}^n$  are Voronoi element centers  $(V_{T_{ev}^1}, V_{T_{ev}^2}, \dots, V_{T_{ev}^K})$  (see [Figure 87](#) – [Figure 89](#)).

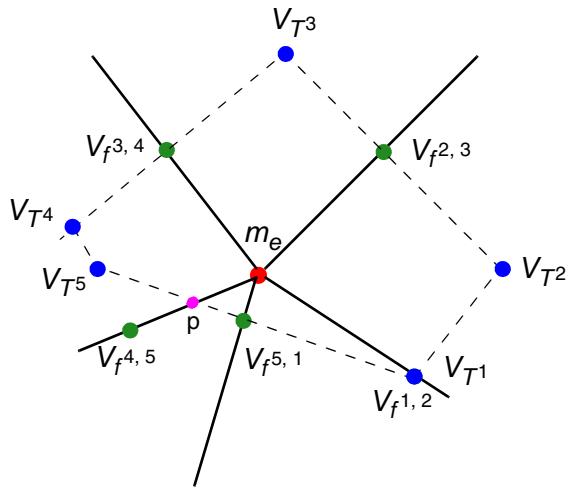
## Chapter 38: Numeric Methods

### Box Method Coefficients in the 3D Case

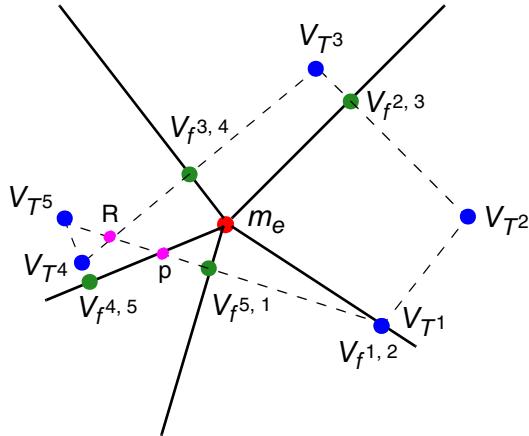
**Figure 87** Face of the Voronoï box for Delaunay mesh without obtuse elements: view of mid-perpendicular plane at edge e with Voronoï element centers  $V_{T^i}$  and the Voronoï face center  $V_{f^{i,i+1}}$  between elements  $T^i$  and  $T^{i+1}$



**Figure 88** Element 5 is an obtuse element; the face of the Voronoï box is a polygon  $(V_{T^1}, V_{T^2}, V_{T^3}, V_{T^4}, V_{T^5}, V_{T^1})$ , and all vertices are Voronoï element centers



**Figure 89** Element 5 is a non-Delaunay element; the face of the Voronoï box is a polygon  $(V_{T^1}, V_{T^2}, V_{T^3}, R, V_{T^1})$ , and vertex  $R$  is not a Voronoï element center



### Element Intersection Box Method Algorithm

Sentaurus Device uses the element intersection box method algorithm. Let  $S_{ev}^{T^i}$  be the area of intersection  $F_{ev} \cap T^i$ . For example:

1. Not obtuse elements (see [Figure 87](#)) or elements  $T^1, T^2, T^3$  in (see [Figure 88](#) and [Figure 89](#)):

$$S_{ev}^{T^i} = \text{Area}(m_e, V_{f^{i-1,i}}, V_{T^i}, V_{f^{i,i+1}}, m_e)$$

2. Obtuse elements (see [Figure 88](#)):

$$S_{ev}^{T^4} = \text{Area}(m_e, V_{f^{3,4}}, V_{T^4}, V_{T^5}, p, m_e) \text{ and } S_{ev}^{T^5} = \text{Area}(m_e, p, V_{f^{5,1}}, m_e)$$

3. Non-Delaunay elements (see [Figure 89](#)):

$$S_{ev}^{T^4} = \text{Area}(m_e, V_{f^{3,4}}, R, p, m_e) \text{ and } S_{ev}^{T^5} = \text{Area}(m_e, p, V_{f^{5,1}}, m_e)$$

Let edge  $e$  have vertices  $v1, v2$ . If all elements around this edge are Delaunay elements, then  $S_{ev1}^{T^i} = S_{ev2}^{T^i}$ . For non-Delaunay elements,  $S_{ev1}^{T^i} \neq S_{ev2}^{T^i}$ .

The parameters needed for discretization,  $\mu(\Omega_i^e)$  and  $\kappa_{ij}^e$  from [Equation 1266](#), are 2D arrays  $\mu(T^i, v)$  and  $\kappa(T^i, e)$  ( $v \in T^i$  is a vertex of the element, and  $e = e(v1, v2)$  is the edge of the element).

## Chapter 38: Numeric Methods

### Box Method Coefficients in the 3D Case

The options for computing the box method coefficients are:

- AverageBoxMethod

$$\kappa(T^i, e) = 0.5 \cdot (S_{ev1}^{T^i} + S_{ev2}^{T^i}) / (\text{length}(e)) \quad (1267)$$

For non-Delaunay elements, you have the average coefficient value.

- NaturalBoxMethod

$$\kappa(T^i, e) = (S_{ev1}^{T^i}) / (\text{length}(e)) \quad (1268)$$

This algorithm has no averaging.

Both algorithms have the same coefficients for the Delaunay mesh. Only one box method algorithm can be activated. After computing the box method coefficients, Sentaurus Device uses these values for computation control volumes  $\mu(T^i, v)$  using standard analytic formulas.

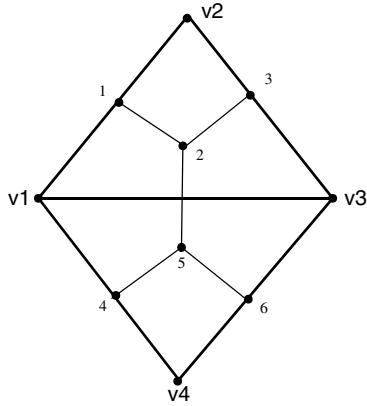
## Truncated Obtuse Elements

If a mesh has no obtuse elements, you have element-volume conservation for `Measure` values (see [Figure 90](#)):

$$\text{Vol}(T^i) = \mu(T^i, v) \quad (1269)$$

$v \in T^i$

*Figure 90 Element-volume conservation for mesh without obtuse elements*



For a Delaunay mesh, you have total-volume conservation (see [Figure 91](#)):

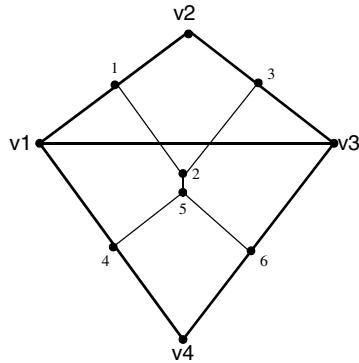
$$V \equiv \text{Vol}(T^i) = \mu(T^i, v) \equiv V_{\text{BM}} \quad (1270)$$

$i \quad i \quad v \in T^i$

## Chapter 38: Numeric Methods

### Box Method Coefficients in the 3D Case

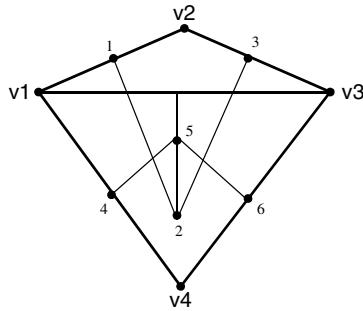
*Figure 91 Total-volume conservation for Delaunay mesh*



For a non-Delaunay mesh, you have no even total-volume conservation (see [Figure 92](#)):

$$\delta V = \text{abs}(V - V_{\text{BM}}) > 0 \quad (1271)$$

*Figure 92 Violation of total-volume conservation for non-Delaunay mesh*



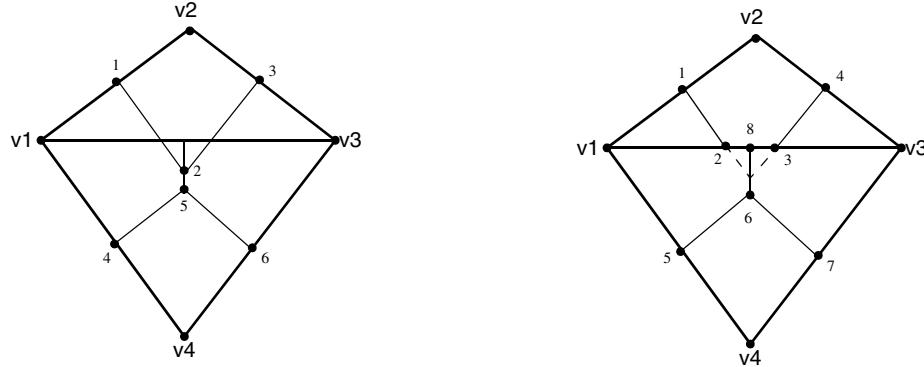
There are problems for which element-volume conservation is very important (such as optical electronic or diffusion in Sentaurus Process). For these operations, Sentaurus Device has the special option `MixAverageBoxMethod`.

In this case, Sentaurus Device uses `AverageBoxMethod` to compute the coefficients and the algorithm *truncation obtuse elements* to compute the control volumes. [Figure 93](#) shows the difference between the original and truncated Voronoï polygons in the 2D case.

## Chapter 38: Numeric Methods

### Box Method Coefficients in the 3D Case

**Figure 93** Algorithm truncation obtuse element: (left) Voronoï polygons before truncation –  $P1(v1,1,2,5,4,v1)$ ,  $P2(v2,1,2,3,v2)$ ,  $P3(v3,3,2,5,6,v3)$ ,  $P4(v4,4,5,6,v4)$ ; (right) Voronoï polygons after truncation –  $P1(v1,1,2,8,6,5,v1)$ ,  $P2(v2,1,2,3,4,v2)$ ,  $P3(v3,4,3,8,6,7,v3)$ ,  $P4(v4,5,6,7,v4)$



For the 3D case, a similar algorithm of truncation is used.

[Table 211 on page 1573](#) lists all the available options for computing box method parameters.

## Weighted Box Method Coefficients

The main goal of any space discretization is the generation of a Delaunay mesh. In this case, the box method coefficients are positive and the finite volume scheme [1] ([Equation 1266](#)) is monotone. For a non-Delaunay mesh, the `AverageBoxMethod` coefficients are positive, but the order of the approximation of PDEs is less than one.

Sentaurus Mesh can use special technology – Delaunay–Voronoi weights – for which the *weighted Voronoï diagram* has no overlap control volume for a non-Delaunay mesh. As a result, the finite volume scheme is monotone and the order of the approximation PDEs is equal to one.

## Weighted Points

A weighted point  $\tilde{p} = (p, P^2)$  is interpreted as a sphere (circle in two dimensions) with a center  $p$  and radius  $P$ . The weighted distance between  $\tilde{p}$  and  $x = (x, X)$  is defined as [4] [5][6]:

$$\|\tilde{p} - \tilde{x}\| = \sqrt{\|p - x\|^2 - P^2 - X^2} \quad (1272)$$

The weighted points  $\tilde{p}$  and  $\tilde{x}$  are orthogonal if the weighted distance vanishes:  
 $\|\tilde{p} - \tilde{x}\| = 0$ .

In the 3D case, any four weighted points have a common orthogonal sphere called an *orthosphere*. Unless the four centers lie in a common plane, the orthosphere is unique and has a finite radius.

In the 2D case, any three weighted points have a common circle called an *orthocircle*. Unless the three centers lie in a common line, the orthocircle is unique and has a finite radius (see [Figure 94](#)).

*Figure 94 Since the radii of all weighted vertices are positive, their centers lie outside the orthocircle*

## Weighted Voronoï Diagram

The weighted generalization of the Voronoï diagram is obtained by substituting a weighted vertex for vertices and an orthosphere (orthocircle) for circumspheres (circumcircles).

The weighted bisector plane  $B_{ij}$  between  $\tilde{p}_i$  and  $\tilde{p}_j$  is the locus of points at an equal-weighted distance from  $\tilde{p}_i$  and  $\tilde{p}_j$ . The center of the orthosphere  $x$  is the intersection of the bisector planes  $x = \bigcap_{i \neq j} B_{ij}$ . The weighted middle point  $m_{ij}$  between  $\tilde{p}_i$  and  $\tilde{p}_j$  is the intersection of the segment  $[p_i, p_j]$  and the bisector plane  $B_{ij}$ . The value  $m_{ij}$  is equal to:

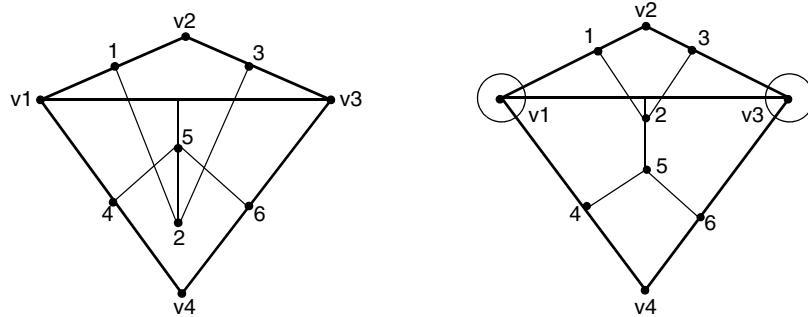
$$m_{ij} = \alpha_i p_i + \alpha_j p_j \quad (1273)$$

where:

$$\alpha_i = 0.5 \cdot \frac{P_i^2 - P_j^2}{\|p_i - p_j\|^2}, \alpha_j = 1 - \alpha_i \quad (1274)$$

Therefore, the weighted bisector plane  $B_{ij}$  is orthogonal to the segment  $[p_i, p_j]$  and contains the weighted middle point  $m_{ij}$ , which is sufficient to compute the weighted coefficients and control volumes. If the radius  $P_i \neq P_j$ , then the middle point  $m_{ij} \neq 0.5(p_i + p_j)$  and the weighted Voronoï diagram have no overlap control volume for non-Delaunay meshes (see [Figure 95](#)). This is the main property of the weighted Voronoï diagram.

**Figure 95** Two-dimensional non-Delaunay mesh: (left) not weighted Voronoï diagram has overlap elements and (right) weighted Voronoï diagram has no overlap elements



Sentaurus Process computes the squared radii ( $P_i^2$ , plot name: `De1VorWeight` [ $\mu\text{m}^2$ ]) of the weighted vertices and writes them to a TDR file (see the option `StoreDelaunayWeight` in the *Sentaurus™ Process User Guide*, Table 90).

If you specify the keyword `WeightedVoronoiBox` in the `Math` section of the command file, Sentaurus Device reads the corresponding arrays from a TDR file and computes the weighted coefficients and measure.

## Saving and Restoring Box Method Coefficients

Usually, the coefficients needed for discretization are computed inside Sentaurus Device. For experimental purposes, it might be preferred to use externally provided data. Measure and Coefficients ( $\mu^e(\Omega_i)$  and  $\kappa_{ij}^e$  from [Equation 1266](#)) can be stored in, and loaded into and from the debug file. There are two options for element numbering in such files:

- Internal Sentaurus Device numbering with `MeasureCoefficientsDebug` as the debug file name
- Mesh numbering (from grid file) with `MeasureCoefficients.debug` as the debug file name for this option

If you specify the keyword `BoxMeasureFromFile` or `BoxCoefficientsFromFile` in the `Math` section and there is the file `MeasureCoefficientsDebug` in the simulation directory, Sentaurus Device reads the corresponding arrays from this file.

If you specify either the keyword `BoxMeasureFromFile(GrdNumbering)` or the keyword `BoxCoefficientsFromFile(GrdNumbering)` and there is the `MeasureCoefficients.debug` file, then Sentaurus Device reads the corresponding arrays from this file. If there are no such debug files but these keywords are specified, then Sentaurus Device computes `Measure` and `Coefficients` and writes them in the corresponding file.

## Chapter 38: Numeric Methods

### Box Method Coefficients in the 3D Case

The format of the `MeasureCoefficientsDebug` file is as follows. In line  $k$  of the `Measure` section, the control volume for each element-vertex  $j$  of element  $k$  is stored (that is, value `Measure[k][j]`). The enumeration of elements and local numbering of vertices inside the element (see [Figure 98 on page 1249](#)) correspond to the internal Sentaurus Device numbering.

The `Coefficients` section in this file has a similar format. For example, the `Measure` section in the file can appear as follows:

```
Measure {
    8.79666833501378e-08 4.359833416750702e-08 4.359833416750729e-08
    8.719666833501378e-08 4.359833416750702e-08 4.359833416750729e-08
    ...
}
```

The format of the `MeasureCoefficients.debug` file differs. There are four sections: `Info`, `Elem_type`, `Measure`, and `Coefficients`. In line  $k$  of the `Measure` section, the control volume for each element-vertex  $j$  of element  $k$  is stored (that is, value `Measure[k][j]`). The enumeration of elements and local numbering of vertices inside the element correspond to the grid file. The `Coefficients` section in the debug file has a similar format.

For example, the file can look like:

```
Info {
    dimension      = 2
    nb_vertices    = 10
    nb_grd_elements = 11
    nb_des_elements = 7
}

Elem_type {
    point        = 0
    line         = 1
    triangle     = 2
    rectangle    = 3
    tetrahedron  = 5
    pyramid      = 6
    prism        = 7
    cuboid       = 8
}

Measure { # unit = [um^2]
# grd_elem des_elem elem_type
0   0     2   1.828427124999999e+00 9.142135625000004e-01
9.142135625000002e-01
1   1     2   4.052251462735666e+00 4.052251462735666e+00
8.104502925471332e+00
...
7   -1    1      # contact or interface
8   -1    1      # contact or interface
...
```

## Chapter 38: Numeric Methods

### Box Method Coefficients in the 3D Case

```
}
```

```
          Coefficients { # unit = [1]
# grd_elem  des_elem  elem_type
0      0      2      1.093836321204215e+00 1.100111438811216e-16
2.285533906249999e-01
1      1      2      0.000000000000000e+00 8.379715512271076e-01
8.379715512271076e-01
...
7      -1      1      # contact or interface
8      -1      1      # contact or interface
...
}
```

---

## Statistics About Non-Delaunay Elements

The log file contains information about region non-Delaunay elements and interface non-Delaunay elements. For details, see *Utilities User Guide*, Chapter 4.

### Region Non-Delaunay Elements

A log file contains common data about the mesh and information about non-Delaunay elements per region (for Delaunay mesh `DeltaVolume=0` and non-Delaunay `Volume=0`):

```
----- Region non-Delaunay elements -----
Region Volume BoxMethodVolume DeltaVolume Elements non-Delaunay
non-DelaunayVolume
name           [um2]           [um2]       [%] Elements           [um2]       [%]
-----
Nitride 1.9500000e-04 2.2635574e-04 16.080 53 12 (22.64 %) 1.8215e-04
(1.1e-05)
...
Oxide   6.0618645e-03 8.0705629e-03 33.137 2500 818 (32.72 %) 2.3715e-04
(2.0e-04)
Silicon 3.5548100e-02 4.9531996e-02 39.338 12656 5057 (39.96 %)
1.0715e-04 (1.0e-05)
Total    4.6402113e-02 6.4934852e-02 39.939 16550 6383 (38.57 %)
2.9218e-04 (2.1e-05)
\-----
!!!!!! WARNING: Region_0 DeltaVolume = 39.338% > 1.0e-04%
```

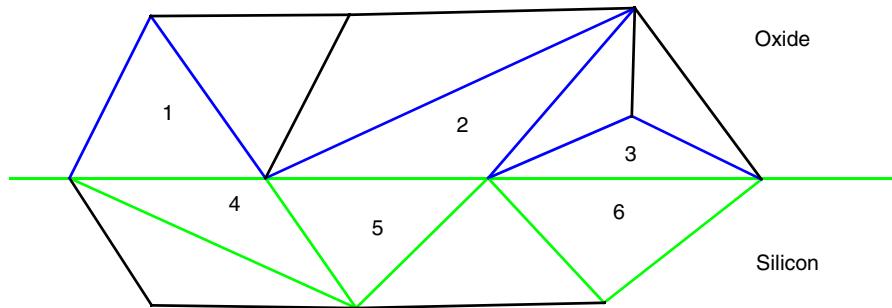
If `DeltaVolume` exceeds some user-defined limit (default:  $10^{-4}$ ), then Sentaurus Device prints a warning message. You can change this limit value in the global `Math` section. For example:

```
Math {
  ...
  DeltaVolumeLimit = 1e-2
}
```

## Interface Non-Delaunay Elements

An *interface element* is an element that has a face (or an edge in two dimensions) lying on the interface. A non-Delaunay element is an *interface non-Delaunay element* only if its obtuse face lies on the surface of the interface (see [Figure 96](#)).

**Figure 96** Blue (1, 2, 3) and green (4, 5, 6) elements are oxide and silicon interface elements, respectively. Elements 2 and 4 are non-Delaunay elements, not interface non-Delaunay elements. Only element 3 is an interface non-Delaunay element.



The following is an example of a log file for interface non-Delaunay elements:

```
/----- Interface non Delaunay elements -----
Region1    Elements      non Delaunay      Volume      non Delaunay
Region2    Elements      Elements        [um2]       DeltaVolume [um2]
-----
..... .
silicon     3           0 (0.00 %)   1.5775139e-03  0.00000e+00 (0.00 %)
oxide       3           1 (33.0 %)   1.6776069e-03  0.11000e-03 (0.10 %)
..... .
Total       6           1 (16.0 %)   3.6951838e-02  0.11000e+00 (0.05 %)
\-----
```

## Plot Section

[Table 189](#) lists the plot variables that might be useful for visualizing box method statistics (see [Scalar Data on page 1516](#)). See *Utilities User Guide*, Chapter 4, for the definitions of these variables.

**Table 189** Plot variables for box method data

Plot variable	Location
BM_AngleElements	Element
BM_CoeffIntersectionNonDelaunayElements	Element
BM_ElementsWithCommonObtuseFace	Element

## Chapter 38: Numeric Methods

### AC Simulation

Table 189 Plot variables for box method data (Continued)

Plot variable	Location
BM_ElementsWithObtuseFaceOnBoundaryDevice	Element
BM_ElementVolume	Element
BM_IntersectionNonDelaunayElements	Element
BM_VolumeIntersectionNonDelaunayElements	Element
BM_wCoeffIntersectionNonDelaunayElements	Element
BM_wElementsWithCommonObtuseFace	Element
BM_wElementsWithObtuseFaceOnBoundaryDevice	Element
BM_wIntersectionNonDelaunayElements	Element
BM_wVolumeIntersectionNonDelaunayElements	Element
BM_AngleVertex	Vertex
BM_EdgesPerVertex	Vertex
BM_ElementsPerVertex	Vertex
BM_ShortestEdge	Vertex

---

## Improved Accuracy of Box Method Parameters

For challenging meshes containing slivers or non-Delaunay elements, it is recommended to use the `BM_StableCalculation` option in the global `Math` section of the command file. It is switched off by default.

Specifying this option activates enhancements to the box method calculations, which lead to the improved accuracy of the resulting parameters. Consequently, the `DeltaVolume` quality measure is reduced for non-Delaunay meshes.

---

## AC Simulation

AC simulation is based on small-signal AC analysis. The response of the device to ‘small’ sinusoidal signals superimposed upon an established DC bias is computed as a function of

## Chapter 38: Numeric Methods

### AC Simulation

frequency and the DC operating point. Steady-state solution is used to build up a linear algebraic system [7] whose solution provides the real and imaginary parts of the variation of the solution vector ( $\phi, n, p, T_n, T_p, T$ ) induced by small sinusoidal perturbation at the contacts.

## AC Response

The AC response is obtained from the three basic semiconductor equations (see [Equation 37 on page 229](#) and [Equation 57 on page 238](#)) and from up to three additional energy conservation equations to account for electron, hole, and lattice temperature responses. In the following description of the AC system, temperatures have been omitted in the solution vector and Jacobian for simplicity, a complete description being formally obtained by adding the temperature responses to the solution vector and the corresponding lines to the system Jacobian.

After discretization, the simplified system of equations can be symbolically represented at node  $i$  of the computation mesh as:

$$F_{\phi i}(\phi, n, p) = 0 \quad (1275)$$

$$\dot{F}_{ni}(\phi, n, p) = \dot{G}_{ni}(n) \quad (1276)$$

$$\dot{F}_{pi}(\phi, n, p) = \dot{G}_{pi}(p) \quad (1277)$$

where  $F$  and  $G$  are nonlinear functions of the vector arguments  $\phi, n, p$ , and the dot denotes time differentiation.

By substituting the vector functions of the form  $\xi_{\text{total}} = \xi_{\text{DC}} + \tilde{\xi}e^{i\omega t}$  into [Equation 1275](#), [Equation 1276](#), and [Equation 1277](#) where  $\xi = \phi, n, p$ ,  $\xi_{\text{DC}}$  is the value of  $\xi$  at the DC operating point, and  $\tilde{\xi}$  is the corresponding response (or the phasor uniquely identifying the complex perturbation) and then expanding the nonlinear functions  $F$  and  $G$  in the Taylor series around the DC operating point and keeping only the first-order terms (the small-signal approximation), the AC system of equations at node  $i$  can be written as:

$$j \begin{bmatrix} \frac{\partial F_{\phi i}}{\partial \phi_j} & \frac{\partial F_{\phi i}}{\partial n_j} & \frac{\partial F_{\phi i}}{\partial p_j} \\ \frac{\partial F_{ni}}{\partial \phi_j} \frac{\partial F_{ni}}{\partial n_j} - i\omega \frac{\partial G_{ni}}{\partial n_j} & \frac{\partial F_{ni}}{\partial p_j} \\ \frac{\partial F_{pi}}{\partial \phi_j} & \frac{\partial F_{pi}}{\partial n_j} & \frac{\partial F_{pi}}{\partial p_j} - i\omega \frac{\partial G_{pi}}{\partial p_j} \end{bmatrix}_{\text{DC}} \begin{bmatrix} \tilde{\phi}_j \\ \tilde{n}_j \\ \tilde{p}_j \end{bmatrix} = 0 \quad (1278)$$

where the solution vector is scaled with respect to terminal voltages (at the contact where the voltage is applied,  $\phi$  is 1). Therefore, the unit of carrier density responses is  $\text{cm}^{-3}\text{V}^{-1}$  and the potential response is unitless.

## Chapter 38: Numeric Methods

### AC Simulation

The matrix of [Equation 1278](#) differs from the Jacobian of the system of equations [Equation 1275](#), [Equation 1276](#), and [Equation 1277](#) only by pure imaginary additive terms involving derivatives of  $G$  with respect to carrier densities.

The global AC matrix system is obtained by imposing the corresponding AC boundary conditions and performing the summation (assembling the global matrix).

Common AC boundary conditions used in AC simulation are Neumann boundary and oxide–semiconductor jump conditions carried over directly from DC simulation; Dirichlet boundary conditions for carrier densities where  $n$  and  $p$  at Ohmic contacts are  $\tilde{n} = \tilde{p} = 0$ ; and Dirichlet boundary conditions for AC potential at Ohmic contacts that are used to excite the system.

After assembling the global AC matrix and taking into account the boundary conditions, the AC system becomes:

$$[J + iD]\tilde{X} = B \quad (1279)$$

where:

- $J$  is the Jacobian matrix.
- $D$  contains the contributions of the  $G$  functions to the matrix.
- $B$  is a real vector dependent on the AC voltage drive.
- $\tilde{X}$  is the AC solution vector.

By writing the solution vector as  $\tilde{X} = X_R + iX_I$  with  $X_R$  and  $X_I$  the real and imaginary part of the solution vector, respectively, the AC system can be rewritten using only real arithmetic as:

$$\begin{bmatrix} J & -D \\ D & J \end{bmatrix} \begin{bmatrix} X_R \\ X_I \end{bmatrix} = \begin{bmatrix} B \\ 0 \end{bmatrix} \quad (1280)$$

The AC response is actually computed by solving the system [Equation 1279](#) or [Equation 1280](#).

An `ACPlot` statement in the `ACCoupled` statement is used to plot AC responses  $\phi, n, p, T_n, T_p, T$ . Responses are plotted in the AC plot file of Sentaurus Device with a separate file for each frequency.

For details about the `ACPlot` statement, see [Table 338 on page 1717](#). For details and examples of small-signal AC analysis, see [Small-Signal AC Analysis on page 149](#).

## AC Current Density Responses

When the AC system is solved, the AC current density responses  $\overset{\circ}{J}_D$ ,  $\overset{\circ}{J}_n$ , and  $\overset{\circ}{J}_p$  are computed using:

$$\overset{\circ}{J}_D = -i\omega\epsilon\nabla\phi \quad (1281)$$

$$\overset{\circ}{J}_n = \frac{\partial \overset{\circ}{J}_n}{\partial \phi} \Big|_{DC} \tilde{\phi} + \frac{\partial \overset{\circ}{J}_n}{\partial n} \Big|_{DC} \tilde{n} + \frac{\partial \overset{\circ}{J}_n}{\partial p} \Big|_{DC} \tilde{p} \quad (1282)$$

$$\overset{\circ}{J}_p = \frac{\partial \overset{\circ}{J}_p}{\partial \phi} \Big|_{DC} \tilde{\phi} + \frac{\partial \overset{\circ}{J}_p}{\partial p} \Big|_{DC} \tilde{p} + \frac{\partial \overset{\circ}{J}_p}{\partial n} \Big|_{DC} \tilde{n} \quad (1283)$$

The unit of current density responses is  $A\text{cm}^{-2}\text{V}^{-1}$ .

The responses of the heat fluxes for the lattice ( $S_L$ ), electrons ( $S_n$ ), and holes ( $S_p$ ) are analogous. Their unit is  $\text{Wcm}^{-2}\text{V}^{-1}$ .

The `ACplot` statement in the `System` section is used to plot the AC current density responses. Responses are added to the AC solution response in the AC plot files of Sentaurus Device.

---

## Harmonic Balance Analysis

Harmonic balance (HB) analysis is a frequency domain method to solve periodic and quasi-periodic time-dependent problems for steady-state solutions [8][9]. It is a popular method for RF circuit design applications. While transient discretization schemes allow the simulation of arbitrary time-dependent problems, HB more efficiently models periodic and quasi-periodic problems for systems with time constants that vary by many orders of magnitude. The detailed command file syntax is given [Harmonic Balance on page 153](#).

---

## Harmonic Balance Equation

In general, the dynamic mixed-mode simulation problem takes the form:

$$\frac{d}{dt}q[r, u(t, r)] + f[r, u(t, r), w(t)] = 0 \quad (1284)$$

where  $f$  and  $q$  are nonlinear functions,  $w$  represents explicitly time-dependent devices (in particular, voltage or current sources), and the function  $u$  is the vector of all solution variables.

## Chapter 38: Numeric Methods

### Harmonic Balance Analysis

Let  $f_1, \dots, f_K$  be a set of different frequencies with  $f_k > 0$ , then both the sources and the solution are approximated by a truncated Fourier series:

$$u(t) = U_0 + \sum_{k=1}^K \{ U_k \exp(i\omega_k t) + U_k^* \exp(-i\omega_k t) \} \quad (1285)$$

A formal Fourier transform of [Equation 1284](#) results in the HB equation for the problem:

$$L(U) = i\Omega Q(U) + F(U) = 0 \quad (1286)$$

where  $F$  and  $Q$  are the finite Fourier series of  $f$  and  $q$ , respectively,  $\Omega$  is the frequency matrix, and  $U$  is the vector of all Fourier coefficients of  $u$ .

## Multitone Harmonic Balance Analysis

Multitone HB analysis makes use of the multidimensional Fourier transformation (MDFT). This means that the problem is mapped onto a problem in a multidimensional frequency and multidimensional time domain, thereby exploiting the equivalence of Fourier spectra of quasi-periodic functions with their corresponding multidimensional functions.

## Multidimensional Fourier Transformation

The MDFT maps multidimensional functions onto a multidimensional spectrum.

Let  $M$  be a positive integer, the number of tones,  $f_1, \dots, f_M$ , be a finite set of different positive numbers, the base frequencies of tones, and  $H_1, \dots, H_M$  nonnegative integer numbers, the maximal number of harmonics for each tone.

Define for each tone  $m$  the  $m$ -th base period  $T_m := 1/f_m$ , the  $m$ -th circular frequency  $\omega_m := 2\pi f_m$ , the  $m$ -th (minimum) number of sampling points  $S_m := 2H_m + 1$ , and the  $m$ -th (maximum) sampling interval  $\delta_m = T_m/S_m$ .

Furthermore, let  $\underline{x} = (x_1, \dots, x_M)^T$  denote the  $M$ -dimensional vector, and let  $D_{\underline{x}} = \text{diag}(x_1, \dots, x_M)$  denote the  $M \times M$ -matrix composed of the values  $x_1, \dots, x_M$ .

The set of multi-indices associated with  $\underline{H}$  is given by:

$$K := \{\underline{h} \in \mathbb{Z}^M : -H_m \leq h_m \leq H_m \text{ for all } 1 \leq m \leq M\} \quad (1287)$$

Let  $u^M$  be a function on the  $M$ -dimensional space  $\mathbb{C}^M$  given by:

$$u^M(t_1, \dots, t_M) = \sum_{\underline{h} \in K} U_{\underline{h}}^M \exp(i\underline{h} D_{\underline{\omega}} \underline{t}) \quad (1288)$$

## Chapter 38: Numeric Methods

### Harmonic Balance Analysis

with given complex numbers  $U_{\underline{h}}^M$ , then:

$$U_{\underline{h}}^M = \frac{1}{T} \sum_{t=0}^{T_1} \dots \sum_{t=M}^{T_M} u^M(t_1, \dots, t_M) \exp(-i\underline{h}D_{\underline{\omega}}t) dt_1 \dots dt_M \quad (1289)$$

with  $T := \prod_{1 \leq m \leq M} T_m$ .  $U^M = (U_{\underline{h}}^M)_{\underline{h} \in K}$  is the multidimensional spectrum of  $u^M$ .

Sampling the function in all dimensions at the equidistant sampling points  $t_s = D_{\underline{s}}S (0 \leq s < S)$ , the discrete MDFT is written formally as:

$$U^M = \Gamma^M u^M \quad (1290)$$

that is,  $\Gamma^M$  is a linear map from  $C^S$  onto  $C^S$  where  $S := \prod_{1 \leq m \leq M} S_m$  is the total number of sampling points.

## Quasi-Periodic Functions

The multidimensional function  $u^M$  can be projected onto a one-dimensional time space by:

$$u(t) := u^M(t, \dots, t) = \sum_{\underline{h} \in K} U_{\underline{h}} \exp(i\underline{h}\omega t) \quad (1291)$$

Functions satisfying this representation are called quasi-periodic. The set:

$$\Lambda := \{f_{\underline{h}} \in \mathbb{R} : f_{\underline{h}} = \underline{h} \cdot \hat{f} \text{ for all } \underline{h} \in K\} \quad (1292)$$

is the spectrum domain associated with  $f$  and  $K$  (or  $H$ ). The projection is invertible if, for two different multi-indices  $\underline{h}_1$  and  $\underline{h}_2$  in  $K$ , the resulting frequencies  $f_{\underline{h}_1}$  and  $f_{\underline{h}_2}$  differ. Note that the one-dimensional Fourier spectrum of  $u$  coincides with the multidimensional spectrum of  $u^M$ .

While for the multidimensional function  $u^M$   $S$  sample points can be specified to compute the multidimensional spectrum, the one-dimensional sample points for  $u$  are not well defined (but are virtual in the multidimensional time domain).

## Multidimensional Frequency Domain Problem

Multitone HB analysis is essentially a translation of (one-dimensional or multidimensional) time-domain problems in a multidimensional frequency domain. Although originally derived from a time-domain problem, circuit equations are specified directly in a multidimensional frequency domain. This avoids sampling of (one-dimensional) time-dependent sources, which cannot be performed accurately on a sample set of size  $S$ . This is the reason why the compact circuit models must provide the CMI-HB-MDFT function set.

The Fourier transformation of quasi-periodic functions is the composition:

$$\Gamma = \Gamma^M \circ P^{-1} \quad (1293)$$

where  $\Gamma^M$  is the multidimensional Fourier transformation of [Equation 1290](#) and  $P^{-1}$  is the inverse of the projection [Equation 1291](#).

## One-Tone Harmonic Balance Analysis

For one-tone HB analysis, the standard discrete Fourier transformation can be used, which includes that the sampling points are defined explicitly in a (one-dimensional) time domain. Therefore, the problem can be extracted directly from the time-domain formulation of the circuit.

## Solving the Harmonic Balance Equation

The HB equation ([Equation 1286](#)) is a nonlinear equation in  $U$  and is solved by the Newton algorithm. In each Newton step, the following linear equation must be solved:

$$\frac{\partial L}{\partial U}(U) \cdot \delta U = -L(U) \quad (1294)$$

The Jacobian  $\partial L / \partial U$  in Fourier space is computed from the Jacobian in the time domain as follows. For a nonlinear scalar function  $g: \mathbb{R} \rightarrow \mathbb{R}$  and a  $T$ -periodic scalar signal  $u(t)$ , the Fourier coefficients  $G \in \mathbb{C}^S$  of  $g(u(t))$  are approximated:

$$G(U) = \Gamma g(\hat{u}) = \Gamma g(\Gamma^{-1}U) \quad (1295)$$

where  $\Gamma$  and  $\Gamma^{-1}$  are the discrete Fourier transform operator and its inverse,  $\hat{u}$  is the vector of the time samples  $u(t_i)$ , and  $g(u)$  is the vector of values  $g(u(t_i))$ .

The derivatives of the  $k$ -th Fourier component  $G_k$  with respect to the  $j$ -th Fourier component  $U_j$  read:

$$\frac{\partial G_k}{\partial U_j}(U) = \sum_l \Gamma_{kl} \frac{\partial g}{\partial U_j}(\hat{u}_l) = \sum_l \Gamma_{kl} \frac{\partial g}{\partial u}(\hat{u}_l) \Gamma_{lj}^{-1} \quad (1296)$$

The corresponding Jacobian is written in the compact form:

$$\frac{\partial G}{\partial U} = \hat{\Gamma} J_u \Gamma^{-1} \text{ with } \hat{\Gamma} = \sum_l \frac{\partial g}{\partial u}(\hat{u}_l) \quad (1297)$$

For scalar functions  $g$  and  $u$ ,  $J_u$  is a diagonal matrix. For vector-valued functions  $g$  and  $u$ ,  $J_u$  is a block-diagonal matrix.

Using the notation above, [Equation 1286](#) becomes:

$$L(U) = i\Omega \Gamma q(\hat{u}(U)) + \Gamma f(\hat{u}(U)) = 0 \quad (1298)$$

## Chapter 38: Numeric Methods

### Harmonic Balance Analysis

and [Equation 1294](#) for the Newton step takes the form:

$$(i\Omega \hat{\Gamma} \hat{J}_q \hat{\Gamma}^{-1} + \hat{\Gamma} \hat{J}_f \hat{\Gamma}^{-1}) \delta U = -L(U) \quad (1299)$$

The Newton algorithm constructs a sequence  $U^k$  of the Fourier coefficients of the time-domain solution vector  $u$ . The sequence is considered to be converged if both the residual  $|L(U^k)|$  and the update error are small.

## Solving the Harmonic Balance Newton Step Equation

The memory requirements for storing the HB Jacobian matrix typically become very large, as its size is increased by a factor of  $S^2$  compared to the corresponding DC or transient matrix. For a very small number of harmonics and a moderately sized simulation grid, using a direct linear solver might be feasible. However, using the GMRES(m) iterative method is recommended for most applications.

## Restarted GMRES Method

The HB module makes use of a preconditioned restarted generalized minimum residual GMRES(m) method [\[10\]](#), a Krylov subspace method, which does not need to store the Jacobian in memory, as only matrix-vector products have to be computed.

GMRES(m) requires a suitable preconditioner to achieve convergence. A (left) preconditioner  $P$  is a matrix that approximates a given matrix  $A$ , but is much easier to invert than  $A$  itself. Instead of solving the linear equation  $Ax = b$  for given  $A$  and  $b$ , the (left) preconditioned problem  $P^{-1}Ax = P^{-1}b$  is solved. The preconditioner used for HB [\[11\]](#) takes the form:

$$P = i\Omega \begin{bmatrix} \bar{J}_q & 0 \\ \dots & \dots \\ 0 & \bar{J}_q \end{bmatrix} + \begin{bmatrix} \bar{J}_f & 0 \\ \dots & \dots \\ 0 & \bar{J}_f \end{bmatrix} \quad (1300)$$

where the matrix  $\bar{J}_f$ , and similarly  $\bar{J}_q$ , is computed as:

$$\bar{J}_f = \frac{1}{S} \sum_{0 \leq s \leq S-1} J_f(t_s) \quad (1301)$$

and  $J_f$  denotes the Jacobian of  $f$  with respect to  $u$ . The preconditioner equals the HB Jacobian in the limit of small signals, where the coupling terms between frequencies vanish. Therefore, each diagonal block of  $P$  corresponds to the AC matrix for the respective harmonic. This preconditioner is well suited to moderately large signal applications.

The preconditioner can be computed without an explicit Fourier transform, and its inversion is more economical than for the full Jacobian. The inversion is performed by applying a complex direct solver for each harmonic component separately, thereby requiring the computational costs of solving  $(S+1)/2$  complex-valued linear systems. The

## Chapter 38: Numeric Methods

### Transient Simulation

computational complexity is of the order  $O(S)$  for inverting the preconditioner, and  $O(S \ln(S))$  for one complete iteration step of the iterative solver, while the number of iterations necessary to achieve convergence is unknown (but bounded).

## Direct Solver Method

For the direct solver, the complex-valued  $S \times S$  linear system ([Equation 1299](#)) is transformed to a  $S \times S$  real-valued problem, which is possible as only real-valued functions are involved. The resulting linear system is solved by the direct solver PARDISO. The direct solver requires the entire matrix stored in memory. Therefore, the memory capacity is easily exceeded for increasing  $S$ . In addition, the computational complexity is of the order  $O(S^3)$ .

---

## Transient Simulation

Transient equations used in semiconductor device models and circuit analysis can be formally written as a set of ordinary differential equations:

$$\frac{d}{dt}q(z(t)) + f(t, z(t)) = 0 \quad (1302)$$

which can be mapped to the DC and transient parts of the PDEs.

Sentaurus Device uses implicit discretization of transient equations (see [Equation 1302](#)) and supports two discretization schemes: simple backward Euler and composite trapezoidal rule/backward differentiation formula (TRBDF), which is the default.

---

## Backward Euler Method

Backward Euler is a very stable method, but it has only a first-order of approximation over time step  $h_n$ . The discretization can be written as:

$$q(t_n + h_n) + h_n f(t_n + h_n) = q(t_n) \quad (1303)$$

The local truncation error (LTE) estimation is based on the comparison of the obtained solution  $q(t_n + h_n)$  with the linear extrapolation from the previous time step.

The extrapolated solution is written as:

$$q^{\text{extr}} = q(t_n) - \frac{f(t_n) + f(t_n + h_n)}{2} h_n \quad (1304)$$

Then, in every point, the relative error can be estimated as  $(q(t_n + h_n) - q^{\text{extr}})/q(t_n + h_n)$ .

## Chapter 38: Numeric Methods

### Transient Simulation

Using [Equation 1303](#) and [Equation 1304](#), and estimating the norm of relative error, Sentaurus Device computes the value:

$$r = \sqrt{\frac{1}{N} \sum_{i=1}^N \frac{|f(t_n + h_n) - f(t_n)|}{\varepsilon_{R,tr} |q_n(t_n + h_n)| + \varepsilon_{A,tr}} h_n^2} \quad (1305)$$

where the sum is taken over all unknowns (that is, all free vertices of all equations), and  $\varepsilon_{R,tr}$  and  $\varepsilon_{A,tr}$  are the relative and absolute transient errors, respectively.

The next time step is estimated as:

$$h_{\text{est}} = h_n r^{-1/2} \quad (1306)$$

The value of the estimated time step is used for the  $h_{n+1}$  computation (see [Controlling Transient Simulations on page 1201](#)).

## TRBDF Composite Method

The transient scheme [12] for the approximation of [Equation 1302](#) is briefly reviewed here. From each time point  $t_n$ , the next time point  $t_n + h_n$  ( $h_n$  is the current step size) is not directly reached. Instead, a step in between to  $t_n + \gamma h_n$  is made. This improves the accuracy of the method, and  $\gamma = 2 - \sqrt{2}$  has been shown to be the optimal value. Using this, two nonlinear systems are reached.

For the trapezoidal rule (TR) step:

$$2q(t_n + \gamma h_n) + \gamma h_n f(t_n + \gamma h_n) = 2q(t_n) - \gamma h_n f(t_n) \quad (1307)$$

For the BDF2 step:

$$(2 - \gamma)q(t_n + h_n) + (1 - \gamma)h_n f(t_n + h_n) = (1/\gamma)(q(t_n + \gamma h_n) - (1 - \gamma)^2 q(t_n)) \quad (1308)$$

The LTE is estimated after such a double step as:

$$\tau = \left[ \frac{f(t_n)}{\gamma} - \frac{f(t_n + \gamma h_n)}{\gamma(1 - \gamma)} + \frac{f(t_n + h_n)}{1 - \gamma} \right] \quad (1309)$$

$$C = \frac{-3\gamma^2 + 4\gamma - 2}{12(2 - \gamma)} \quad (1310)$$

Sentaurus Device then computes the following value from this:

$$r = \sqrt{\frac{1}{N} \sum_{i=1}^N \frac{\tau_i}{\varepsilon_{R,tr} |q_n(t_n + h_n)| + \varepsilon_{A,tr}}} \quad (1311)$$

where the sum is taken over all unknowns (that is, all free vertices of all equations), and  $\varepsilon_{R,tr}$  and  $\varepsilon_{A,tr}$  are the relative and absolute transient errors, respectively.

## Chapter 38: Numeric Methods

### Transient Simulation

Since the TRBDF method has a second-order approximation over  $h_n$ , the next step can be estimated as:

$$h_{\text{est}} = h_n r^{-1/3} \quad (1312)$$

The value of the estimated time step is used for the  $h_{n+1}$  computation (see [Controlling Transient Simulations on page 1201](#)).

---

## Controlling Transient Simulations

By default, Sentaurus Device uses the TRBDF method. To switch to backward Euler (BE), you can specify `Transient=BE` either in the `Math` section (which changes the default time integration method) or in the options of a `Transient` statement in the `Solve` section (which changes the time integration method only for that particular `Transient` statement).

The TRBDF method is a second-order integration method that provides better accuracy than the backward Euler method, and is the only reliable method for challenging transient problems. Alternatively, the often recommended backward Euler method is a first-order integration method, which provides faster and more robust convergence behavior than the TRBDF method, but at the expense of accuracy, often filtering out fast-varying signals such as ringing.

To evaluate whether a time step was successful and to provide an estimate for the next step size, the following rules are applied:

- If one of the nonlinear systems cannot be solved, then the step is refused and tried again with  $h_n = 0.5 \cdot h_n$ .
- Otherwise, the inequality  $r < 2f_{\text{rej}}$  is tested. If it is fulfilled, then the transient simulation proceeds with  $h_{n+1} = h_{\text{est}}$ . Otherwise, the step is tried again with  $h_n = 0.9 \cdot h_{\text{est}}$ .
- The LTE is checked only if the `CheckTransientError` option is selected. Otherwise, the selection of the next time step is based only on convergence of nonlinear iterations.

To activate LTE evaluation and time-step control, you must specify `CheckTransientError` either globally (in the `Math` section) or locally as an option in the `Transient` statement. The keyword `NoCheckTransientError` deactivates time-step control. The value of the relative error is defined by the parameter `TransientDigits` according to [Equation 14 on page 139](#).

The absolute error is given by the keyword `TransientError` or is recomputed from `TransientErrRef` ( $x_{\text{ref,tr}}$ ) using [Equation 15 on page 140](#) (if `RelErrControl` is switched on). Sentaurus Device provides the default values of  $\varepsilon_{R,\text{tr}}$ ,  $\varepsilon_{A,\text{tr}}$ , and  $x_{\text{ref,tr}}$ . The coefficient  $f_{\text{rej}}$  is equal to 1 by default. You can define the values of  $\varepsilon_{R,\text{tr}}$ ,  $\varepsilon_{A,\text{tr}}$ ,  $x_{\text{ref,tr}}$ , and  $f_{\text{rej}}$  globally in the `Math` section, or you can specify them as options in the `Transient` statement. In the latter case, it overwrites the default and `Math` specifications for this command.

## Chapter 38: Numeric Methods

### Transient Simulation

## Floating Gates

During a transient time step, the charge  $Q$  of a floating gate is updated as a function of the injection current  $i$ :

$$\Delta Q = \int_{t_1}^{t_2} i(t) dt \quad (1313)$$

$\Delta Q$  represents the charge increase for the time step  $[t_1, t_2]$ .

Because the floating-gate charge is not updated self-consistently during a transient step, but only as a postprocessing operation, the numeric update is given by:

$$\Delta Q = i(t_1) \cdot \Delta t \quad (1314)$$

where  $\Delta t = t_2 - t_1$ . The error of this numeric approximation is estimated by:

$$\Delta Q_{\text{error}} = \frac{|i(t_2) - i(t_1)|}{2} \Delta t \quad (1315)$$

With the option `CheckTransientError`, the error  $\Delta Q_{\text{error}}$  in the charge update is monitored as well. In the case of relative error control (`RelErrControl`), a transient step is accepted only if the following condition holds:

$$\frac{\Delta Q_{\text{error}}}{10^{-\text{Digits}} (|Q| + \text{ErrRef})} < 1 \quad (1316)$$

The values of `Digits` and `ErrRef` can be specified in the `Math` section. For example:

```
Math {
    TransientDigits = 3
    TransientErrRef (Charge) = 1.602192e-19
}
```

In the case of absolute error control (`-RelErrControl`), a transient step is accepted only if the following condition holds:

$$\frac{\Delta Q_{\text{error}}}{\frac{q}{10^{-\text{Digits}} |Q|} + \text{Error}} < 1 \quad (1317)$$

The electron charge  $q = 1.602192 \cdot 10^{-19}$  C is used as a scaling factor.

The values of `Digits` and `Error` can be specified in the `Math` section. For example:

```
Math {
    TransientDigits = 3
    TransientError (Charge) = 1e-3
}
```

## Chapter 38: Numeric Methods

### Nonlinear Solvers

Note that the values of `ErrRef` and `Error` are related by the equation:

$$\text{ErrRef} = \frac{\text{Error}}{10^{-\text{Digits}}} q \quad (1318)$$

---

## Nonlinear Solvers

In the next sections, the `Digits` variable corresponds to the keyword `Digits`, which can be given in the `Math` section (see [Convergence and Error Control on page 193](#)), or in parentheses of each `Plugin` or `Coupled` statement.

---

## Fully Coupled Solution

For the solution of nonlinear systems, the scheme developed by Bank and Rose [13] is applied. This scheme tries to solve the nonlinear system  $\mathbf{g}(z) = 0$  by the Newton method:

$$\mathbf{\hat{g}} + \mathbf{\hat{g}}' \mathbf{\hat{x}} = 0 \quad (1319)$$

$$\mathbf{\hat{z}}^j - \mathbf{\hat{z}}^{j+1} = \lambda \mathbf{\hat{x}} \quad (1320)$$

where  $\lambda$  is selected such that  $\|\mathbf{\hat{g}}_{k+1}\| < \|\mathbf{\hat{g}}_k\|$ , but is as close as possible to 1. Sentaurus Device handles the error by computing an error function that can be defined by two methods.

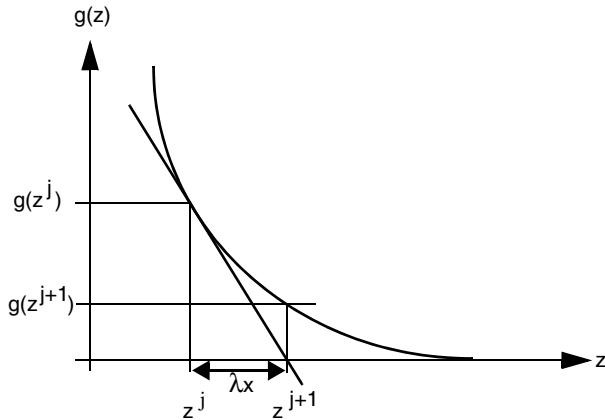
The Newton iterations stop if the convergence criteria are fulfilled. One convergence criterion is the norm of the right-hand side, that is,  $\|\mathbf{\hat{g}}\|$  in [Equation 1319](#). Another natural criterion can be the relative error of the variables measured, such as  $\left\| \frac{\lambda \mathbf{\hat{x}}}{\mathbf{\hat{z}}} \right\|$ .

Conversely, for very small  $\mathbf{\hat{z}}$  updates,  $\lambda \mathbf{\hat{x}}$  must be measured with respect to some reference value of the variable  $z_{\text{ref}}$ . The formula used in Sentaurus Device as the second convergence criterion is:

$$\frac{1}{\varepsilon_R N} \frac{|z(e, i, j) - z(e, i, j-1)|}{|z(e, i, j)| + z_{\text{ref}}(e)} < 1 \quad (1321)$$

where  $z(e, i, j)$  is the solution of the equation  $e$  (Poisson, electron, hole, and so on) at node  $i$  after Newton iteration  $j$ . The constant  $N$  is given by the total number of nodes multiplied by the total number of equations. The parameter  $\varepsilon_R$  is the relative error criterion.

Figure 97 Newton iteration



The value of  $\varepsilon_R = 10^{-\text{Digits}}$  is set by specifying the following in the Math section:

```
Math{...
    Digits = 5
}
```

where 5 is the default for Digits. The reference values  $z_{\text{ref}}(e)$  ensure numeric stability even for cases when  $z(e, i, j)$  is zero or very small. This error condition ensures that the respective equations are solved to an accuracy of approximately  $z_{\text{ref}}(e)\varepsilon_R$ .

Equation 1321 can be written in the symbolic form:

$$\frac{1}{\varepsilon_R} \left\| \frac{\lambda x}{z^j + z_{\text{ref}}} \right\| < 1 \quad (1322)$$

Equation 1322 can also be rewritten in the equivalent form:

$$\left\| \frac{\lambda \bar{x}}{\varepsilon_R \bar{z}^j + \varepsilon_A} \right\| < 1 \quad (1323)$$

where  $\bar{z}^j = z^j/z^*$  and  $\bar{x} = x/z^*$ .

$z^*$  is the normalization factor (for example, it is the intrinsic carrier density  $n_i = 1.48 \times 10^{10} \text{ cm}^{-3}$  for electron and hole equations, and the thermal voltage  $u_{T0} = 25.8 \text{ mV}$  for the Poisson equation).

The absolute error is related to the relative error through:

$$\varepsilon_A = \varepsilon_R \frac{z_{\text{ref}}}{z^*} \quad (1324)$$

## Chapter 38: Numeric Methods

### Nonlinear Solvers

Sentaurus Device supports two schemes for controlling error conditions. The default scheme is based on [Equation 1321](#). The default values for the parameters  $z_{\text{ref}}$  are listed in [Table 342 on page 1722](#). They also are accessible in the `Math` section.

For example:

```
Math{...
    ErrRef( Electron ) = 1e10
    ErrRef( Hole )     = 1e10
}
```

The second scheme is activated with the keyword `-RelErrControl` in the `Math` section and is based on [Equation 1323](#). The default values for the parameters  $\varepsilon_A$  are listed in [Table 342](#). They also are accessible in the `Math` section. For example:

```
Math{...
    -RelErrControl
    Error( Electron ) = 1e-5
    Error( Hole )     = 1e-5
}
```

---

## ‘Plugin’ Iterations

This is the traditional scheme, which is also known as *Gummel iterations* in most other device simulators. Consider that there are  $n$  sets of nonlinear systems  $g_j(z_1 \dots z_n) = 0$  ( $n$  can be, for example, 3 and the sets can be the Poisson equation and two continuity equations). This method starts with values  $z_1^{(1)}, \dots, z_n^{(1)}$  and then solves each set  $g_j = 0$  separately and consecutively.

One loop could be:

$$\begin{aligned} g_1(z_1 z_2^{(i)} \dots z_n^{(i)}) &= 0 & z_1^{(i+1)} \\ \dots \\ g_1(z_1^{(i+1)} \dots z_{n-1}^{(i+1)} z_n) &= 0 & z_n^{(i+1)} \end{aligned} \tag{1325}$$

If an update  $(\lambda x)$  of the solution between two successive plugin iterations is defined as:

$$(\lambda x) = z_j^{(i+1)} - z_j^{(i)} \tag{1326}$$

[Equation 1322](#) or [Equation 1323](#) can be applied for convergence control in plugin iterations.

## References

- [1] R. E. Bank, D. J. Rose, and W. Fichtner, "Numerical Methods for Semiconductor Device Simulation," *IEEE Transactions on Electron Devices*, vol. ED-30, no. 9, pp. 1031–1041, 1983.
- [2] R. S. Varga, *Matrix Iterative Analysis*, Englewood Cliffs, New Jersey: Prentice-Hall, 1962.
- [3] E. M. Buturla *et al.*, "Finite-Element Analysis of Semiconductor Devices: The FIELDAY Program," *IBM Journal of Research and Development*, vol. 25, no. 4, pp. 218–231, 1981.
- [4] H. Edelsbrunner, "Triangulations and meshes in computational geometry," *Acta Numerica*, vol. 9, pp. 133–213, March 2000.
- [5] S.-W. Cheng *et al.*, "Sliver Exudation," *Journal of the ACM*, vol. 47, no. 5, pp. 883–904, 2000.
- [6] H. Edelsbrunner and D. Guoy, "An Experimental Study of Sliver Exudation," in *Proceedings of the 10th International Meshing Roundtable*, Newport Beach, CA, USA, pp. 307–316, October 2001.
- [7] S. E. Laux, "Application of Sinusoidal Steady-State Analysis to Numerical Device Simulation," in *New Problems and New Solutions for Device and Process Modelling: An International Short Course held in association with the NASECODE IV Conference*, Dublin, Ireland, pp. 60–71, 1985.
- [8] B. Troyanovsky, Z. Yu, and R. W. Dutton, "Physics-based simulation of nonlinear distortion in semiconductor devices using the harmonic balance method," *Computer Methods in Applied Mechanics and Engineering*, vol. 181, no. 4, pp. 467–482, 2000.
- [9] P. J. C. Rodrigues, *Computer-Aided Analysis of Nonlinear Microwave Circuits*, Boston: Artech House, 1998.
- [10] Y. Saad, *Iterative Methods for Sparse Linear Systems*, Philadelphia: SIAM, 2nd ed., 2003.
- [11] P. Feldmann, B. Melville, and D. Long, "Efficient Frequency Domain Analysis of Large Nonlinear Analog Circuits," in *Proceedings of the IEEE Custom Integrated Circuits Conference*, San Diego, CA, USA, pp. 461–464, May 1996.
- [12] R. E. Bank *et al.*, "Transient Simulation of Silicon Devices and Circuits," *IEEE Transactions on Computer-Aided Design*, vol. CAD-4, no. 4, pp. 436–451, 1985.
- [13] R. E. Bank and D. J. Rose, "Global Approximate Newton Methods," *Numerische Mathematik*, vol. 37, no. 2, pp. 279–295, 1981.

# 39

## Physical Model Interface

---

*This chapter discusses the flexible physical model interface (PMI) that is used to add new physical models to Sentaurus Device.*

---

### Introduction to the Physical Model Interface

The PMI provides direct access to certain models in the semiconductor transport equations. You can provide new C++ functions to compute these models, and Sentaurus Device loads the functions at runtime using the dynamic loader. No access to the Sentaurus Device source code is necessary.

You can modify the models in the following sections:

- [Generation–Recombination](#)
  - [Avalanche Generation](#)
  - [Generation–Recombination](#)
  - [Lifetimes](#)
  - [Nonlocal Generation–Recombination](#)
  - [Tunneling Parameters](#)
- [Mobility](#)
  - [Doping-Dependent Mobility](#)
  - [Fitting Parameter for Ballistic Mobility](#)
  - [High-Field Saturation](#)
  - [High-Field Saturation With Two Driving Forces](#)
  - [Mobility Degradation at Interfaces](#)
- [Semiconductor Band Structure](#)
  - [Apparent Band-Edge Shift](#)

## **Chapter 39: Physical Model Interface**

Introduction to the Physical Model Interface

- Band Gap
- Bandgap Narrowing
- Effective Mass
- Electron Affinity
- Phase and State Transitions
  - Multistate Configuration–Dependent Apparent Band-Edge Shift
  - Multistate Configuration–Dependent Bulk Mobility
  - Multistate Configuration–Dependent Heat Capacity
  - Multistate Configuration–Dependent Thermal Conductivity
- Thermal Properties and Heat
  - Distributed Thermal Resistance
  - Heat Capacity
  - Heat Generation Rate
  - Metal Thermoelectric Power
  - Thermal Conductivity
  - Thermoelectric Power
- Optics
  - Complex Refractive Index Model Interface
  - Optical Quantum Yield
  - Special Contact PMI for Raytracing
- Mechanical Stress
  - Mobility Stress Factor
  - Piezoelectric Polarization
  - Piezoresistive Coefficients
  - Stress
- Traps and Fixed Charges
  - Trap Capture and Emission Rates
  - Trap Energy Shift

## **Chapter 39: Physical Model Interface**

Introduction to the Physical Model Interface

- Degradation
  - Diffusivity
  - eNMP Transition Rates
- Ferroelectrics and Ferromagnetics
  - Ferroelectrics
  - Ferroelectrics Hysteresis
  - Ferromagnetism and Spin Transport
- Electrical Resistivity
  - Metal Resistivity
  - Schottky Resistance
- Simulation Controls
  - Current Plot File
  - Postprocessing for Transient Simulations
  - Preprocessing for Newton Iterations and Newton Step Control
- Various
  - Dielectric Permittivity
  - Energy Relaxation Times
  - Gamma Factor for Density Gradient Model
  - Heavy Ion Spatial Distribution
  - Hot-Carrier Injection
  - Incomplete Ionization
  - Space Factor

---

## Using a PMI Model

To use a PMI model in a Sentaurus Device simulation, the following steps are required:

1. Implement a C++ subroutine to evaluate the PMI model. In the case of the standard interface, you must write additional C++ subroutines to evaluate the derivatives of the PMI model with respect to all input variables.
2. Compile the C++ subroutine that implements the PMI model with `cmi <pmi_file_name>.c`. This produces the `<pmi_file_name>.so.<platform>` shared object file that Sentaurus Device loads at runtime (see [Shared Object Code on page 1230](#)).

**Note:**

The version of the C++ compiler used for a PMI model must be identical to the version of the C++ compiler used to compile Sentaurus Device. Use the command `cmi -a` to verify the compiler versions.

3. Define the `PMIPath` variable in the `File` section of the command file. This variable defines the search path for the shared object files. A PMI model is activated in the `Physics` section of the command file by specifying its name (see [Command File of Sentaurus Device on page 1230](#)).

Parameters for PMI models can appear in the parameter file under a section with the PMI model name (see [Parameter File of Sentaurus Device on page 1232](#)). Alternatively, you can also define the PMI model parameters in the Sentaurus Device command file that uses the PMI model (see [Command File of Sentaurus Device on page 1230](#)).

The source code for the examples presented in this chapter is located in the following directory:

```
$STROOT/tcad/$STRELEASE/Applications_Library/GettingStarted/sdevice/pmi
```

---

## Available Interfaces

For most models, Sentaurus Device provides two equivalent interfaces:

- The *standard C++ interface* is based on the data type `double`. Separate subroutines must be written to evaluate the model and its derivatives. This interface provides performance comparable to the built-in models in Sentaurus Device (see [Standard C++ Interface](#)).
- The *simplified C++ interface* is based on the data type `pmi_float`. Only a single subroutine must be implemented to evaluate the model (see [Simplified C++ Interface on page 1214](#)). For local models, the derivatives of the model are obtained by automatic differentiation. This interface also supports extended-precision floating-point arithmetic (see [Extended Precision on page 223](#)).

## Chapter 39: Physical Model Interface

Introduction to the Physical Model Interface

There is an additional interface where values are computed based on nonlocal values of the input variables (see [Nonlocal Interface on page 1218](#)).

### Standard C++ Interface

For each PMI model, you must implement a C++ subroutine to evaluate the model. Additional subroutines are necessary to evaluate the derivatives of the model with respect to all the input variables. More specifically, you must implement a C++ class that is derived from a base class declared in the header file `PMIModels.h`. In addition, a *virtual constructor function* must be provided, which allocates an instance of the derived class.

#### Example: Implementing Auger Recombination as New PMI Model

Consider the implementation of Auger recombination as a new PMI model. (The built-in Auger recombination model is discussed in [Auger Recombination Model on page 489](#).) In its simplest form, Auger recombination can be written as:

$$R_{\text{net}} = C \cdot (n + p) \cdot (np - n_{i,\text{eff}}^2) \quad (1327)$$

Sentaurus Device needs to evaluate the value of  $R_{\text{net}}$  and the derivatives:

$$\begin{aligned} \frac{\partial R_{\text{net}}}{\partial n} &= C(np - n_{i,\text{eff}}^2 + (n + p)p) \\ \frac{\partial R_{\text{net}}}{\partial p} &= C(np - n_{i,\text{eff}}^2 + (n + p)n) \\ \frac{\partial R_{\text{net}}}{\partial n_{i,\text{eff}}} &= -2C(n + p)n_{i,\text{eff}} \end{aligned} \quad (1328)$$

In the header file `PMIModels.h`, the following base class is defined for recombination models:

```
class PMI_Recombination : public PMI_Vertex_Interface {

    public:
        PMI_Recombination (const PMI_Environment& env);
        virtual ~PMI_Recombination () ;

        virtual void Compute_r
            (const double t, const double n, const double p,
             const double nie, const double f, double& r) = 0;

        virtual void Compute_drdt
            (const double t, const double n, const double p,
             const double nie, const double f, double& drdt) = 0;

        virtual void Compute_drdn
            (const double t, const double n, const double p,
             const double nie, const double f, double& drdn) = 0;

        virtual void Compute_drdp
}
```

## Chapter 39: Physical Model Interface

### Introduction to the Physical Model Interface

```
(const double t, const double n, const double p,
const double nie, const double f, double& drdp) = 0;

virtual void Compute_drdnie
(const double t, const double n, const double p,
const double nie, const double f, double& drdnie) = 0;

virtual void Compute_drdf
(const double t, const double n, const double p,
const double nie, const double f, double& drdf) = 0;
};
```

To implement a PMI model for Auger recombination, you must declare a derived class:

```
#include "PMIModels.h"

class Auger_Recombination : public PMI_Recombination {

    double C;

public:
    Auger_Recombination (const PMI_Environment& env);
    ~Auger_Recombination ();

    void Compute_r
    (const double t, const double n, const double p,
    const double nie, const double f, double& r);

    void Compute_drdt
    (const double t, const double n, const double p,
    const double nie, const double f, double& drdt);

    void Compute_drdn
    (const double t, const double n, const double p,
    const double nie, const double f, double& drdn);

    void Compute_drdp
    (const double t, const double n, const double p,
    const double nie, const double f, double& drdp);

    void Compute_drdnie
    (const double t, const double n, const double p,
    const double nie, const double f, double& drdnie);

    void Compute_drdf
    (const double t, const double n, const double p,
    const double nie, const double f, double& drdf);
};
```

The constructor of the derived class is invoked for each region of the device.

## Chapter 39: Physical Model Interface

### Introduction to the Physical Model Interface

In this example, the variable `c` is initialized from the parameter file:

```
Auger_Recombination::  
Auger_Recombination (const PMI_Environment& env) :  
    PMI_Recombination (env)  
{ C = InitParameter ("C", 1e-30);  
}
```

If the parameter `C` is not found in the parameter file, then a default value of  $10^{-30}$  is used (see [Parameter File of Sentaurus Device on page 1232](#)). During a Newton iteration, Sentaurus Device evaluates a PMI model for each mesh vertex. The method `Compute_r()` computes the recombination rate for a given vertex. According to the parameter list, the recombination rate can depend on the following variables:

- `t`: Lattice temperature
- `n`: Electron density
- `p`: Hole density
- `nie`: Effective intrinsic density
- `f`: Absolute value of electric field

The result of the function is stored in the parameter `r`:

```
void Auger_Recombination::  
Compute_r (const double t, const double n, const double p,  
          const double nie, const double f, double& r)  
{ r = C * (n + p) * (n*p - nie*nie);  
    if (r < 0.0) {  
        r = 0.0;  
    }  
}
```

Besides `Compute_r()`, you must implement other methods to compute the partial derivatives of the recombination rate with respect to the input variables `t`, `n`, `p`, `nie`, and `f`.

The implementation of `Compute_drdn()` to compute the value of  $\partial R / \partial n$  is:

```
void Auger_Recombination::  
Compute_drdn (const double t, const double n, const double p,  
const double nie, const double f, double& drdn)  
{ double r = C * (n + p) * (n*p - nie*nie);  
    if (r < 0.0) {  
        drdn = 0.0;  
    } else {  
        drdn = C * ((n*p - nie*nie) + (n + p) * p);  
    }  
}
```

## Chapter 39: Physical Model Interface

Introduction to the Physical Model Interface

Finally, you must provide a virtual constructor function, which allocates a variable of the new class:

```
extern "C"  
PMI_Recombination* new_PMI_Recombination (const PMI_Environment& env)  
{ return new Auger_Recombination (env);  
}
```

**Note:**

This function must have C linkage and exactly the same name as declared in the header file `PMIModels.h`.

## Simplified C++ Interface

PMI models that utilize the simplified C++ interface are referred to as *simplified PMI models*, and they need to implement essentially only one function that computes the numeric value of the quantity of interest at the actual vertex of the simulation mesh. Derivatives of the quantity with respect to input quantities can be extracted automatically.

The following C++ types provide the appropriate interface:

- Simplified PMI models use a special numeric data type `pmi_float` (see [Numeric Data Type `pmi\_float`](#)).
- Simplified PMI models are derived from the `PMI_Vertex_Common_Base` class, providing an interface at the scope of the PMI model (see [Support at Model Scope on page 1237](#)).
- The main function of a simplified PMI model is called `compute` and has the form:

```
void compute ( const Input& input, Output& output )
```

where `Input` is a class derived from the `PMI_Vertex_Input_Base` class, providing runtime support at the `compute` scope (see [Support at Compute Scope on page 1240](#)).

### Numeric Data Type `pmi_float`

The simplified interface is based on the data type `pmi_float`, which behaves similarly to a `double` and supports all of the usual arithmetic operations:

- Assignment:

```
pmi_float x = 2;  
pmi_float y (x);
```

- Unary operators:

```
+x; -y;
```

- Binary operators:

```
x + y; x - y; x * y; x / y;
```

## Chapter 39: Physical Model Interface

### Introduction to the Physical Model Interface

- Comparisons:

```
x == y; x != y; x < y; x <= y; x > y; x >= y;
```

- Mathematical functions:

```
abs(x); acos(x); acosh(x); asin(x); asinh(x); atan(x); atanh(x);
atan2(y,x); cos(x); cosh(x); erf(x); erfc(x); exp(x); expml(x);
hypot(x,y); isinf(x); isnan(x); ldexp(x,exp); log(x); log1p(x);
log10(x); pow(x,y); pow_int(x,n); sin(x); sinh(x); sqrt(x);
tan(x); tanh(x);
```

- Output:

```
std::cout << x;
```

The following static function returns the accuracy of the floating-point arithmetic:

```
pmi_e_precision pmi_float::get_precision()
```

The result is expressed as an enumeration type:

```
enum pmi_e_precision {
    pmi_c_np, // normal precision (double): -ExtendedPrecision
    pmi_c_xp, // extended precision (long double): ExtendedPrecision
    pmi_c_dd, // double-double: ExtendedPrecision(128)
    pmi_c_qd, // quad-double: ExtendedPrecision(256)
    pmi_c_mp // arbitrary precision: ExtendedPrecision(Digits=...)
};
```

Because the class `pmi_float` supports automatic differentiation, it must store both the value of a variable and the gradient vector of the derivatives with respect to the independent variables. The following methods are available to read the value of a variable, the size of its gradient vector, and the components of the gradient vector:

```
template <class des_t_float> des_t_float get_value();
size_t size_gradient();
template <class des_t_float> des_t_float get_gradient(size_t i);
```

Additional methods are available to set the value of a variable or its gradient:

```
template <class des_t_float> void set_value (const des_t_float a);
template <class des_t_float> void set_gradient (size_t i,
                                              const des_t_float a);
```

#### Note:

These methods for reading and writing the value and the gradient of a variable are not necessary for most models. They might be useful in cases where automatic differentiation yields incorrect results.

## Chapter 39: Physical Model Interface

Introduction to the Physical Model Interface

### Pseudo-Implementation of a Simplified PMI Model

Compared to the standard C++ interface, the simplified C++ interface only requires the implementation of a single subroutine to evaluate the model. As in [Example: Implementing Auger Recombination as New PMI Model on page 1211](#), this section discusses how Auger recombination can be implemented as a PMI model.

The header file `PMI.h` defines the following base class for recombination models:

```
class PMI_Recombination_Base : public PMI_Vertex_Base {

public:
    class Input : public PMI_Vertex_Input_Base {
    public:
        pmi_float t;      // lattice temperature
        pmi_float n;      // electron density
        pmi_float p;      // hole density
        pmi_float nie;   // effective intrinsic density
        pmi_float f;      // absolute value of electric field
    };

    class Output {
    public:
        pmi_float r;    // recombination rate
    };

    PMI_Recombination_Base (const PMI_Environment& env);
    virtual ~PMI_Recombination_Base ();

    virtual void compute (const Input& input, Output& output) = 0;
};
```

To implement a user model, you must first declare a derived class:

```
#include "PMI.h"

class Auger_Recombination : public PMI_Recombination_Base {

private:
    double C;

public:
    Auger_Recombination (const PMI_Environment& env);
    ~Auger_Recombination ();

    virtual void compute (const Input& input, Output& output);
};
```

In the constructor, the variable `C` is initialized from the parameter file:

```
Auger_Recombination::
Auger_Recombination (const PMI_Environment& env) :
    PMI_Recombination_Base (env)
```

## Chapter 39: Physical Model Interface

Introduction to the Physical Model Interface

```
{ C = InitParameter ("C", 1e-30);  
}
```

The constructor is called for each region of the device to ensure that regionwise parameters are handled correctly.

Next, the actual `Compute` function must be implemented. It relies on the auxiliary classes `Input` and `Output` to read the input variables, and to store the recombination rate:

```
void Auger_Recombination::  
compute (const Input& input, Output& output)  
  
{ output.r = C * (input.n + input.p) *  
           (input.n*input.p - input.nie*input.nie);  
  if (output.r < 0.0) {  
    output.r = 0.0;  
  }  
}
```

Finally, a virtual constructor must be supplied to allocate instances of the class `Auger_Recombination`:

```
extern "C"  
PMI_Recombination_Base* new_PMI_Recombination_Base  
(const PMI_Environment& env)  
  
{ return new Auger_Recombination (env);  
}
```

### Note:

This function must have C linkage and exactly the same name as declared in the header file `PMI.h`.

It is possible to implement a model using both the standard and simplified interfaces within the same file. In this case, Sentaurus Device selects the version based on the floating-point precision:

- The standard interface is selected for normal precision (64 bits), which ensures a performance similar to built-in models.
- The simplified interface is selected for extended precision floating-point arithmetic. No loss of accuracy occurs when the PMI model is invoked.

### Note:

The simplified interface is ideally suited for prototyping a new model. No derivatives need to be implemented, which accelerates the development cycle. After a model has been validated, you can convert it easily into the standard interface for performance-critical applications.

## Chapter 39: Physical Model Interface

### Introduction to the Physical Model Interface

As the simplified interface calculates the derivatives for you, it is easy to overlook cases where the expressions themselves are well defined, but their derivatives are not. Assume, for example, that your model computes  $\sqrt{n - n_0}$ , and you have ensured that  $n - n_0$  cannot become negative. This is not enough because, when  $n = n_0$ , the derivative  $1/2\sqrt{n - n_0}$  becomes infinite.

#### Note:

Ensure that not only the expressions you use, but also their derivatives are always valid.

## Nonlocal Interface

With the nonlocal interface model, values are computed based on nonlocal values of the input variables. For example, the generation–recombination rate in a vertex might depend on the carrier densities observed in a remote vertex.

A nonlocal interface provides more flexibility but, in turn, requires additional functionality in the user PMI code. You must implement separate C++ functions for the following purposes:

- *Dependencies*: The variables on which the model depends must be declared, for example, electrostatic potential or carrier densities.
- *Structure of the Jacobian matrices*: For each input variable, the structure (stencil) of the corresponding Jacobian matrix must be declared. For example:

model value in vertex 17 depends on:  
electrostatic potential in vertex 22  
electron density in vertex 55

model value in vertex 18 depends on:  
hole density in vertex 35  
lattice temperature in vertex 44

- *Model values and their derivatives*: The code must evaluate the model values and their derivatives with respect to all dependencies.
- *Update of Jacobian matrices*: Optionally, the PMI can require an update in the structure of the Jacobian matrices. This can be useful if the model does not have purely geometric dependencies, but depends on the values of the solution as well. In this case, Sentaurus Device calls the PMI to request the updated structures of the Jacobian matrices.

Nonlocal PMIs are available with both the data type `double` (see [Standard C++ Interface on page 1211](#)) and the data type `pmi_float` (see [Simplified C++ Interface on page 1214](#)).

However, the data type `pmi_float` is used only to support extended precision floating-point arithmetic. It is not used for the purpose of automatic differentiation.

## Chapter 39: Physical Model Interface

Introduction to the Physical Model Interface

### Jacobian Matrix

Sentaurus Device provides the classes `des_jacobian` (standard C++ interface) and `sdevice_jacobian` (simplified C++ interface) to represent Jacobian matrices. These classes are used to:

- Define the structure of the Jacobian matrices, that is, the dependencies of the model values on the input variables
- Store the derivatives of the model values with respect to the input variables

The size of a Jacobian matrix depends on the location of the model and the location of the input variable. The number of rows is determined by the location of the model. For example, the number of rows for a vertex-based model is given by the number of mesh vertices.

The supported locations are given by the type `des_data::des_location` (for the standard C++ interface):

```
typedef
enum { vertex, edge, element, rivertex, element_vertex }
des_location;
```

and by the type `sdevice_data::sdevice_location` (for the simplified C++ interface):

```
typedef
enum { vertex, edge, element, rivertex, element_vertex }
sdevice_location;
```

Similarly, the number of columns of a Jacobian matrix is determined by the location of the input variable. For example, the number of columns for an edge-based input variable is given by the number of mesh edges.

For a scalar model depending on a scalar input variable, each Jacobian entry is also a simple scalar. However, Sentaurus Device also supports the general case where the model value, or the input variable, or both are vector quantities. In this case, each entry in the Jacobian matrix becomes a small dense matrix of size number-of-inner-rows multiplied by number-of-inner-columns. The number of inner rows is given by the number of model values (1 for a scalar or the mesh dimension for vectors). Similarly, the number of inner columns is determined by the number of variable values (1 for a scalar or the mesh dimension for vectors).

The class `des_jacobian` provides the following methods:

```
des_jacobian (int rows, int cols, int inner_rows, int inner_cols);

int size_rows () const;
int size_cols () const;
int size_inner_rows () const;
int size_inner_cols () const;
int size_matrix () const;

void define_element (int row, int col);
```

## Chapter 39: Physical Model Interface

### Introduction to the Physical Model Interface

```
double* element (int row, int col);

des_jacobian_iterator begin ();
des_jacobian_iterator end ();
des_jacobian_iterator lower_bound (int row, int col);
des_jacobian_iterator upper_bound (int row, int col);

void set (double value);
```

#### Note:

The class `sdevice_jacobian` for the simplified C++ interface provides the same methods, except that the data type `pmi_float` is used instead of `double`.

The constructor creates a new empty Jacobian matrix with the given dimensions.

The methods `size_rows()` and `size_cols()` return the number of rows and columns, respectively. Similarly, the methods `size_inner_rows()` and `size_inner_cols()` return the number of inner rows and inner columns, respectively. The method `size_matrix()` returns the number of nonzero elements in the Jacobian.

Use the method `define_element()` to define the location of a nonzero matrix element. The value of the nonzero entry remains unspecified at this point. However, the required storage is allocated.

The method `element()` returns a pointer to an entry of the Jacobian matrix. A NULL pointer is returned for a nonexistent entry. The pointer defines the beginning of a dense matrix of size number-of-inner-rows multiplied by number-of-inner-columns. The entries in this matrix are stored in row-major order (C style). This means that the derivative of the  $i$ -th component of the result with respect to the  $j$ -th component of the input variable is stored in the location:

```
element() + i * size_inner_cols() + j
```

The functions `begin()` and `end()` return iterators to traverse the nonzero elements of the Jacobian matrix. A typical loop would be:

```
des_jacobian J;
for (des_jacobian_iterator it = J.begin(); it != J.end(); it++) {
    int row = it.row();
    int col = it.col();
    double* value = it.val();
    // process element (row,col)
}
```

The functions `lower_bound()` and `upper_bound()` provide a way to quickly find a range of nonzero elements. The function `lower_bound()` returns an iterator to the first nonzero element not less than  $(row,col)$  in row-major order. Similarly, `upper_bound()` returns an iterator to the first nonzero element greater than  $(row,col)$  in row-major order. Both functions can return `end()` to indicate a nonexistent element.

## Chapter 39: Physical Model Interface

### Introduction to the Physical Model Interface

The following code fragment visits all nonzero elements in row 25:

```
des_jacobian J;
des_jacobian_iterator it_begin = J.lower_bound (25, 0);
des_jacobian_iterator it_end = J.lower_bound (26, 0);
for (des_jacobian_iterator it = it_begin; it != it_end; it++) {
    int row = it.row();
    int col = it.col();
    double* value = it.val();
    // process element (row,col)
}
```

The method `set()` can be used to initialize all matrix elements with a given value.

### Example: Point-to-Point Tunneling

For a nonlocal generation–recombination model, consider a simple point-to-point tunneling model between two vertices  $v_1$  and  $v_2$ . The model compares the electron and hole quasi-Fermi potentials in these two vertices. If  $\Phi_{n,1} < \Phi_{p,2}$ , then the tunneling rate  $r$  is computed as:

$$r = Ae^{-B(E_{V,2} - E_{C,1})^2} \frac{\Delta}{1 + \Delta} \quad (1329)$$

where  $E_C$  is the conduction band energy,  $E_V$  is the valence band energy, and  $\Delta = \Phi_{p,2} - \Phi_{n,1}$ . Nonlocal transport is modeled by using the tunneling rate  $r$  as an electron recombination rate in vertex 1 and a hole recombination rate in vertex 2.

Similarly, if  $\Phi_{p,1} > \Phi_{n,2}$ , then the tunneling rate  $r$  is computed as:

$$r = Ae^{-B(E_{C,2} - E_{V,1})^2} \frac{\Delta}{1 + \Delta} \quad (1330)$$

where  $\Delta = \Phi_{p,1} - \Phi_{n,2}$ . In this case,  $r$  is used as a hole recombination rate in vertex 1 and as an electron recombination rate in vertex 2.

In the header file `PMIModels.h`, the following base class is defined for nonlocal generation–recombination models:

```
class PMI_NonLocal_Recombination : public PMI_Device_Interface {
public:

    class Input {
public:
        const des_region* region; // all vertices belong to this region
        const std::vector<int>& vertices; // list of vertices
    };

    class Output {
public:
        std::vector<double>& elec; // nonlocal recombination rates
                                    // (electrons)
    };
}
```

## Chapter 39: Physical Model Interface

### Introduction to the Physical Model Interface

```
    std::vector<double>& hole; // nonlocal recombination rates
                                // (holes)
    des_id_to_jacobian_map& J_elec; // derivatives (electrons)
    des_id_to_jacobian_map& J_hole; // derivatives (holes)
};

PMI_NonLocal_Recombination (const PMI_Device_Environment& env);
virtual ~PMI_NonLocal_Recombination ();

virtual void
DefineDependencies (std::vector<des_data::des_id>& dependencies)
= 0;

virtual void DefineJacobians (des_id_to_jacobian_map& J_elec,
                             des_id_to_jacobian_map& J_hole) = 0;

virtual void Compute_parallel (const Input& input, Output& output)
= 0;

virtual bool NeedNewEdges () { return false; }
};
```

To implement the point-to-point tunneling model, you must declare a derived class:

```
#include "PMIModels.h"

class P2P_Recombination : public PMI_NonLocal_Recombination {

private:
    double A, B; // model parameters
    int v1, v2; // vertex 1, vertex 2
    double measure1, measure2; // semiconductor node measures for
                               // vertex 1 and 2

public:
    P2P_Recombination (const PMI_Device_Environment& env);

    void DefineDependencies (std::vector<des_data::des_id>&
                            dependencies);
    void DefineJacobians (des_id_to_jacobian_map& J_elec,
                          des_id_to_jacobian_map& J_hole);
    void Compute_parallel (const PMI_NonLocal_Recombination::Input&
                           input, PMI_NonLocal_Recombination::Output&
                           output);
    bool NeedNewEdges ();
};
```

The constructor of the derived class reads the model parameters and computes the semiconductor node measures for the two vertices  $v_1$  and  $v_2$ :

```
P2P_Recombination::
P2P_Recombination (const PMI_Device_Environment& env) :
    PMI_NonLocal_Recombination (env)
{ A = InitParameter ("A", 1e25);
```

## Chapter 39: Physical Model Interface

### Introduction to the Physical Model Interface

```
B = InitParameter ("B", 50);
v1 = InitParameter ("v1", 0);
v2 = InitParameter ("v2", 1);

const des_mesh* mesh = Mesh ();
des_data* data = Data ();
const double*const* measure = data->ReadMeasure ();

// semiconductor node measure for vertex 1
measure1 = 0;
des_vertex* vertex1 = mesh->vertex (v1);
for (size_t eli = 0; eli < vertex1->size_element (); eli++) {
    des_element* el = vertex1->element (eli);
    if (el->bulk ()->material () == "Silicon") {
        for (size_t vi = 0; vi < el->size_vertex (); vi++) {
            des_vertex* v = el->vertex (vi);
            if (v == vertex1) {
                measure1 += measure [el->index()][vi];
            }
        }
    }
}

// semiconductor node measure for vertex 2
measure2 = 0;
des_vertex* vertex2 = mesh->vertex (v2);
for (size_t eli = 0; eli < vertex2->size_element (); eli++) {
    des_element* el = vertex2->element (eli);
    if (el->bulk ()->material () == "Silicon") {
        for (size_t vi = 0; vi < el->size_vertex (); vi++) {
            des_vertex* v = el->vertex (vi);
            if (v == vertex2) {
                measure2 += measure [el->index()][vi];
            }
        }
    }
}
```

The `InitParameter()` method for reading a parameter is documented in [Runtime Support for Vertex-Based PMI Models on page 1237](#). Similarly, the functions for accessing the device mesh are discussed in [Runtime Support for Mesh-Based PMI Models on page 1245](#).

The semiconductor node measures are used later in the `Compute_parallel()` method. They are necessary to ensure current conservation for arbitrary meshes.

The method `DefineDependencies()` defines the variables that will be read later when `Compute_parallel()` is invoked:

```
void P2P_Recombination::
DefineDependencies (std::vector<des_data::des_id>& dependencies)
{ dependencies.push_back (des_data::des_id (des_data::scalar,
```

## Chapter 39: Physical Model Interface

### Introduction to the Physical Model Interface

```
        des_data::vertex,
        "eQuasiFermiPotential" ));
dependencies.push_back (des_data::des_id (des_data::scalar,
                                         des_data::vertex,
                                         "hQuasiFermiPotential" ));
dependencies.push_back (des_data::des_id (des_data::scalar,
                                         des_data::vertex,
                                         "ConductionBandEnergy" ));
dependencies.push_back (des_data::des_id (des_data::scalar,
                                         des_data::vertex,
                                         "ValenceBandEnergy" ));
}
```

The tables in [Appendix F on page 1515](#) list the variables that are available to all mesh-based PMIs. However, the following restrictions must be observed with regard to nonlocal models:

- Constant fields such as doping concentration, mole fraction concentrations, stress fields, and PMI user fields can be used without restrictions. No dependencies need to be defined in `DefineDependencies()`, and no derivatives need to be computed.
- Nonlocal PMIs can use only certain solution-dependent fields. [Table 190](#) lists the subset of variables that are supported.

*Table 190 Solution-dependent data available to nonlocal PMI models*

Data name	Type	Location	Description
BandGap	scalar	vertex	Intrinsic band gap $E_g$
BandgapNarrowing	scalar	vertex	Bandgap narrowing $E_{\text{bgn}}$
ConductionBandEnergy	scalar	element_ vertex	Conduction band energy $E_C$
		vertex	
eDensity	scalar	vertex	Electron density $n$
eEffectiveStateDensity	scalar	vertex	Conduction band density-of-states (DOS) $N_C$
EffectiveIntrinsicDensity	scalar	vertex	Effective intrinsic density $n_{i,\text{eff}}$

## Chapter 39: Physical Model Interface

Introduction to the Physical Model Interface

*Table 190 Solution-dependent data available to nonlocal PMI models (Continued)*

Data name	Type	Location	Description
ElectricField	scalar	element	Electric field $F$
		vertex	
ElectricField	vector	element	
		vertex	
ElectronAffinity	scalar	vertex	Electron affinity $\chi$
ElectrostaticPotential	scalar	vertex	Electrostatic potential $\phi$
eQuasiFermiPotential	scalar	vertex	Electron quasi-Fermi potential $\Phi_n$
eRelativeEffectiveMass	scalar	vertex	Electron DOS mass $m_n$
eTemperature	scalar	vertex	Electron temperature $T_n$
hDensity	scalar	vertex	Hole density $p$
hEffectiveStateDensity	scalar	vertex	Valence band DOS $N_V$
hQuasiFermiPotential	scalar	vertex	Hole quasi-Fermi potential $\Phi_p$
hRelativeEffectiveMass	scalar	vertex	Hole DOS mass $m_p$
hTemperature	scalar	vertex	Hole temperature $T_p$
InsulatorElectricField	scalar	vertex	Electric field $F$ on insulator
		vector	
IntrinsicDensity	scalar	vertex	Intrinsic density $n_i$
LatticeTemperature	scalar	vertex	Lattice temperature $T$
SemiconductorElectricField	scalar	vertex	Electric field $F$ on semiconductor
		vector	

## Chapter 39: Physical Model Interface

### Introduction to the Physical Model Interface

*Table 190 Solution-dependent data available to nonlocal PMI models (Continued)*

Data name	Type	Location	Description
SemiconductorGradValencebandEnergy	scalar	element	Gradient of valence band energy
			$\nabla E_V$
ValenceBandEnergy	scalar	element_ vertex	Valence band energy $E_V$
			vertex

The method `DefineJacobians()` must perform the following operations:

- Allocate all Jacobian matrices with the correct dimensions (see [Jacobian Matrix on page 1219](#)).
- Define the nonzero elements in all Jacobian matrices. All nonzero derivatives that will be computed later in the method `Compute_parallel()` must be defined at this point. The method `Compute_parallel()` cannot allocate additional nonzero entries.

For the point-to-point tunneling model, the method `DefineJacobians()` would be:

```
void P2P_Recombination::
DefineJacobians (des_id_to_jacobian_map& J_elec,
                 des_id_to_jacobian_map& J_hole)
{ const des_mesh* mesh = Mesh ();
  const int n_vertices = mesh->size_vertex ();

  // allocate Jacobians
  des_jacobian*& J_elec_eQF = J_elec [des_data::des_id
    (des_data::scalar, des_data::vertex, "eQuasiFermiPotential")];
  des_jacobian*& J_elec_hQF = J_elec [des_data::des_id
    (des_data::scalar, des_data::vertex, "hQuasiFermiPotential")];
  des_jacobian*& J_elec_EC = J_elec [des_data::des_id
    (des_data::scalar, des_data::vertex, "ConductionBandEnergy")];
  des_jacobian*& J_elec_EV = J_elec [des_data::des_id
    (des_data::scalar, des_data::vertex, "ValenceBandEnergy")];

  des_jacobian*& J_hole_eQF = J_hole [des_data::des_id
    (des_data::scalar, des_data::vertex, "eQuasiFermiPotential")];
  des_jacobian*& J_hole_hQF = J_hole [des_data::des_id
    (des_data::scalar, des_data::vertex, "hQuasiFermiPotential")];
  des_jacobian*& J_hole_EC = J_hole [des_data::des_id
    (des_data::scalar, des_data::vertex, "ConductionBandEnergy")];
  des_jacobian*& J_hole_EV = J_hole [des_data::des_id
    (des_data::scalar, des_data::vertex, "ValenceBandEnergy")];
```

## Chapter 39: Physical Model Interface

### Introduction to the Physical Model Interface

```
J_elec_eQF = new des_jacobian (n_vertices, n_vertices, 1, 1);
J_elec_hQF = new des_jacobian (n_vertices, n_vertices, 1, 1);
J_elec_EC = new des_jacobian (n_vertices, n_vertices, 1, 1);
J_elec_EV = new des_jacobian (n_vertices, n_vertices, 1, 1);

J_hole_eQF = new des_jacobian (n_vertices, n_vertices, 1, 1);
J_hole_hQF = new des_jacobian (n_vertices, n_vertices, 1, 1);
J_hole_EC = new des_jacobian (n_vertices, n_vertices, 1, 1);
J_hole_EV = new des_jacobian (n_vertices, n_vertices, 1, 1);

// define nonzero entries in Jacobians
J_elec_eQF->define_element (v1, v1);
J_elec_eQF->define_element (v2, v2);

J_elec_hQF->define_element (v1, v2);
J_elec_hQF->define_element (v2, v1);

J_elec_EC->define_element (v1, v1);
J_elec_EC->define_element (v2, v2);

J_elec_EV->define_element (v1, v2);
J_elec_EV->define_element (v2, v1);

J_hole_eQF->define_element (v1, v2);
J_hole_eQF->define_element (v2, v1);

J_hole_hQF->define_element (v1, v1);
J_hole_hQF->define_element (v2, v2);

J_hole_EC->define_element (v1, v2);
J_hole_EC->define_element (v2, v1);

J_hole_EV->define_element (v1, v1);
J_hole_EV->define_element (v2, v2);
}
```

The method `Compute_parallel()` computes the electron and hole recombination rates and their derivatives. It can be called during the parallel assembly in Sentaurus Device.

Therefore, it must be implemented in a thread-safe manner. During each call, only the model values and their derivatives for the vertices appearing in the vector `Input::vertices` must be computed.

The recombination rates are multiplied by the semiconductor node measure of the source vertex. This ensures current conservation for arbitrary meshes.

```
void P2P_Recombination::
Compute_parallel (const PMI_NonLocal_Recombination::Input& input,
PMI_NonLocal_Recombination::Output& output)
{ des_data* data = Data ();

  const double* eQF =
    data->ReadScalar (des_data::vertex, "eQuasiFermiPotential");
```

## Chapter 39: Physical Model Interface

### Introduction to the Physical Model Interface

```
const double* hQF =
    data->ReadScalar (des_data::vertex, "hQuasiFermiPotential");
const double* EC =
    data->ReadScalar (des_data::vertex, "ConductionBandEnergy");
const double* EV =
    data->ReadScalar (des_data::vertex, "ValenceBandEnergy");

des_jacobian* J_elec_eQF = output.J_elec [des_data::des_id
    (des_data::scalar, des_data::vertex, "eQuasiFermiPotential")];
des_jacobian* J_elec_hQF = output.J_elec [des_data::des_id
    (des_data::scalar, des_data::vertex, "hQuasiFermiPotential")];
des_jacobian* J_elec_EC = output.J_elec [des_data::des_id
    (des_data::scalar, des_data::vertex, "ConductionBandEnergy")];
des_jacobian* J_elec_EV = output.J_elec [des_data::des_id
    (des_data::scalar, des_data::vertex, "ValenceBandEnergy")];

des_jacobian* J_hole_eQF = output.J_hole [des_data::des_id
    (des_data::scalar, des_data::vertex, "eQuasiFermiPotential")];
des_jacobian* J_hole_hQF = output.J_hole [des_data::des_id
    (des_data::scalar, des_data::vertex, "hQuasiFermiPotential")];
des_jacobian* J_hole_EC = output.J_hole [des_data::des_id
    (des_data::scalar, des_data::vertex, "ConductionBandEnergy")];
des_jacobian* J_hole_EV = output.J_hole [des_data::des_id
    (des_data::scalar, des_data::vertex, "ValenceBandEnergy")];

// compute electron and hole recombination for vertices in
// input domain
for (size_t vi = 0; vi < input.vertices.size (); vi++) {
    const int v = input.vertices [vi];
    if (v == v1 || v == v2) {
        const int v_other = (v == v1) ? v2 : v1;
        const double weight = (v == v1) ? measure2 : measure1;

        if (eQF [v] < hQF [v_other]) {
            double delta = hQF [v_other] - eQF [v]; // delta > 0
            double rate = A * exp (-B * (EV [v_other] - EC [v]) *
                (EV [v_other] - EC [v])) *
                delta / (1 + delta);
            double elec = weight * rate;

            // electron recombination
            output.elec [v] += elec;

            // derivatives
            double deriv_QF = elec / (delta * (1 + delta));
            double deriv_ECEV = 2 * elec * B * (EV [v_other]
                - EC [v]);
            *J_elec_eQF->element (v, v) -= deriv_QF;
            *J_elec_hQF->element (v, v_other) += deriv_QF;
            *J_elec_EC->element (v, v) += deriv_ECEV;
            *J_elec_EV->element (v, v_other) -= deriv_ECEV;
        }
    }
}
```

## Chapter 39: Physical Model Interface

Introduction to the Physical Model Interface

```
if (hQF [v] > eQF [v_other]) {
    double delta = hQF [v] - eQF [v_other]; // delta > 0
    double rate = A * exp (-B * (EC [v_other] - EV [v]) *
                           (EC [v_other] - EV [v])) *
                  delta / (1 + delta);
    double hole = weight * rate;

    // hole recombination
    output.hole [v] += hole;

    // derivatives
    double deriv_QF = hole / (delta * (1 + delta));
    double deriv_ECEV = 2 * hole * B * (EC [v_other]
                                         - EV [v]);
    *J_hole_eQF->element (v, v_other) -= deriv_QF;
    *J_hole_hQF->element (v, v) += deriv_QF;
    *J_hole_EC->element (v, v_other) -= deriv_ECEV;
    *J_hole_EV->element (v, v) += deriv_ECEV;
}
}
}
}
```

In this example, the dependencies of the model do not change as a function of the solution. Therefore, the method `NeedNewEdges()` simply returns false:

```
bool P2P_Recombination::
NeedNewEdges ()
{ return false; // nonlocal edges do not depend on solution }
```

Finally, you must provide a virtual constructor function that allocates a variable of the new class:

```
extern "C"
PMI_NonLocal_Recombination*
new_PMI_NonLocal_Recombination (const PMI_Device_Environment& env)
{ return new P2P_Recombination (env); }
```

### Note:

This function must have C linkage and exactly the same name as declared in the header file `PMIModels.h`.

The example in this section uses the data type `double` according to the standard C++ interface defined in the header file `PMIModels.h`. Alternatively, the simplified C++ interface defined in the header file `PMI.h` uses the data type `pmi_float` to support extended-precision floating-point arithmetic.

## Chapter 39: Physical Model Interface

Introduction to the Physical Model Interface

You can implement a nonlocal model using both the standard and simplified interfaces in the same file. In this case, Sentaurus Device selects the version based on the floating-point precision:

- The standard interface is selected for normal precision (64 bits). This ensures a performance similar to built-in models.
- The simplified interface is selected for extended-precision floating-point arithmetic. No loss of accuracy occurs when the PMI model is invoked.

## Shared Object Code

Sentaurus Device assumes that the shared object code corresponding to a PMI model can be found in the file `modelname.so.arch`. The base name of this file must be identical to the name of the PMI model. The extension `.arch` depends on the hardware architecture.

The script `cmi`, which is also a part of the CMI, can be used to produce the shared object files (see *Compact Models User Guide*, Runtime Support).

---

## Command File of Sentaurus Device

To load PMI models into Sentaurus Device, the `PMIPath` search path must be defined in the `File` section of the command file. The value of `PMIPath` consists of a sequence of directories. For example:

```
File { PMIPath = ". /home/joe/lib /home/mary/sdevice/lib" }
```

For each PMI model that appears in the `Physics` section, the given directories are searched for a corresponding shared object file `modelname.so.arch`.

The PMI in Sentaurus Device provides access to mesh-based scalar fields specified by you. These fields must be defined on the device grid in a separate TDR (`.tdr`) data file. Up to 300 datasets (`PMIUserField0`, ..., `PMIUserField299`) can be defined.

Sentaurus Device reads the user-defined fields if the corresponding file name is given in the command file:

```
File { PMIUserFields = "fields" }
```

You activate a PMI model in the `Physics` section of the command file by specifying the name of the PMI model in the appropriate part of the `Physics` section. Examples for different types of PMI models are:

- Generation–recombination models:

```
Physics { Recombination (pmi_model_name ...) }
```

## Chapter 39: Physical Model Interface

### Introduction to the Physical Model Interface

- Avalanche generation:

```
Physics { Recombination (Avalanche (pmi_model_name ...)) }
```

- Mobility models:

```
Physics {
    Mobility (
        DopingDependence (pmi_model_name)
        Enormal (pmi_model_name)
        ToCurrentEnormal (pmi_model_name)
        HighFieldSaturation (pmi_model_name driving_force)
    )
}
```

A PMI model name can only consist of alphanumeric characters and underscores (\_). The first character must be either a letter or an underscore. A PMI model name can also be quoted as "model\_name" to avoid conflicts with Sentaurus Device keywords.

All PMI models can be specified regionwise or materialwise. For example:

```
Physics (region = "Region.1") { ... }
```

```
Physics (material = "AlGaAs") { ... }
```

PMI models also recognize parameters in the command file. Usually, command file parameters are listed in parentheses after the model name. For example:

```
Physics {
    Recombination (pmi_model_name (a = 1
                                    b = "string"
                                    c = (1.2 3.4 5.6 7.8)
                                    d = ("red" "blue" "green")))
}
```

For certain models, the syntax differs from the previous example, and this is noted in the documentation (see [Appendix G on page 1558](#)).

#### Note:

PMI model parameters also can be specified in the parameter file (see [Parameter File of Sentaurus Device on page 1232](#)). Parameters in the command file take precedence over parameters in the parameter file.

Certain values of PMI models can be plotted in the Plot section of the command file. The following identifiers are recognized:

- Generation–recombination models:

```
Plot {
    PMIRecombination
    PMIENonLocalRecombination  PMIHNonLocalRecombination
}
```

## Chapter 39: Physical Model Interface

Introduction to the Physical Model Interface

- User-defined fields:

```
Plot { PMIUserField0 PMIUserField1 ... PMIUserField299 }
```

- Piezoelectric polarization:

```
Plot { PE_Polarization/vector PE_Charge }
```

- Metal conductivity (see [Metal Resistivity on page 1431](#)):

```
Plot { MetalConductivity }
```

The current plot PMI can be used to add entries to the current plot file:

```
CurrentPlot { pmi_CurrentPlot }
```

---

## Parameter File of Sentaurus Device

For each PMI model, a corresponding section with the same name can appear in the parameter file:

```
PMI_model_name {
    par1 = value
    par2 = value
    ...
}
```

### Note:

Parameter names consist only of alphanumeric characters and underscores (\_).  
The first character must be either a letter or an underscore.

Parameters can be numbers, or strings, or arrays of numbers or strings:

```
PMI_model_name {
    a = 1
    b = "string"
    c = (1.2 3.4 5.6 7.8)
    d = ("red" "blue" "green")
}
```

### Note:

Arrays must consist of either numbers or strings only. Mixed arrays containing both numbers and strings are not supported. An empty array, such as `c=()`, is considered a numeric array of size 0.

Parameters can be specified regionwise and materialwise:

```
Region = "Region.1" {
    PMI_model_name {
        ...
    }
}
```

## Chapter 39: Physical Model Interface

### Introduction to the Physical Model Interface

```
        }
    }
Material = "AlGaAs" {
    PMI_model_name {
        ...
    }
}
```

You can use the method `ReadParameter()` to obtain the value of a parameter given its name. This method returns a pointer to a variable of type `PMIBaseParam`:

```
const PMIBaseParam* p = ReadParameter ("name of parameter");
```

A `NULL` pointer indicates that the parameter has not been defined in the parameter file. Otherwise, the method `PMIBaseParam::ValueType()` returns the value type (`PMIBaseParam::real` or `PMIBaseParam::string`) of the parameter. Similarly, the method `PMIBaseParam::DataType()` returns the data type (`PMIBaseParam::scalar` or `PMIBaseParam::vector`) of the parameter. In the case of an array parameter, the size of the array can be obtained by:

```
PMIBaseParam::size()
```

The size of scalar parameters is always 1.

Depending on the type of a parameter, its value can be accessed through an assignment statement:

```
double a = *p;           // real, scalar
double b = (*p)[index]; // real, vector
const char* c= *p;       // string, scalar
const char* d = (*p)[index]; // string, vector
```

An error occurs if the type of a parameter is incompatible with the left-hand side in the assignment statement.

In the case of a real scalar parameter, the method `InitParameter()` checks whether the parameter has been specified in the parameter file:

```
double value = InitParameter ("pi", 3.14159);
```

If the parameter has been specified, then the given value is taken. Otherwise, the default value is used.

An alternative version of `InitParameter()` is available for vector-valued parameters:

```
std::vector<double> v;
v.push_back (-1);          // default value
v.push_back (-2);          // default value
InitParameter ("vec", v);
```

If the parameter `vec` is found in the parameter file, then the vector `v` is redefined with the new values. Otherwise, the existing default values in `v` are left unchanged.

## Chapter 39: Physical Model Interface

Introduction to the Physical Model Interface

Variants of these two methods are also available for string parameters:

```
const char* value = InitStringParameter ("scalar string",
                                         "default value");
std::vector<const char*> s;
InitStringParameter ("vector string parameter", s);
```

### Note:

PMI model parameters also can be specified in the command file (see [Command File of Sentaurus Device on page 1230](#)). Parameters in the command file take precedence over parameters in the parameter file.

---

## Parallelization

During a parallel simulation, Sentaurus Device can invoke a PMI model concurrently from different threads. Therefore, the computational functions must be implemented in a thread-safe manner.

The following rules guarantee a thread-safe PMI:

- Do not use global variables.
- Class variables are only modified in the constructor and destructor. Class variables can be read in the computational functions, but they must not be modified.
- Within the computational functions, all temporary variables are allocated as either automatic variables or dynamic variables with the help of the `new` and `delete` operators.

## Thread-Local Storage

The thread-local storage template class `PMI_TLS` provides a mechanism to store data per thread. This can be useful to optimize the runtime performance of a PMI. Consider an example where a large data structure is allocated and deallocated with each `compute` call:

```
class Recombination : public PMI_Recombination_Base {
public:
    ...
    void compute (const Input& input, Output& output)
    {
        BigData data;
        data.initialize ();
        // compute output
    }
};
```

## Chapter 39: Physical Model Interface

### Introduction to the Physical Model Interface

With the help of the template class `PMI_TLS`, a copy of `BigData` is allocated for each thread on demand:

```
class Recombination : public PMI_Recombination_Base {
private:
    PMI_TLS<BigData> Data;

public:
    ...
    void compute (const Input& input, Output& output)
    {
        bool exists;
        BigData& data = Data.local (exists);
        if (!exists) {
            data.initialize ();
        }
        // compute output
    }
};
```

The function call `Data.local(exists)` creates a new instance of `BigData` for each thread if it does not exist already. The argument `exists` is used to check whether `data` has been freshly allocated. In this case, `data` is initialized. Otherwise, it was already initialized in a previous `compute()` call.

The template class `PMI_TLS` is declared as follows:

```
template <typename T> class PMI_TLS {
public:
    T& local (bool& exists);
    T& local ();
    size_t size () const;
    T& operator [] (size_t index);
};
```

Both variants of the method `local()` return a reference to a thread-local element. The optional argument `exists` indicates whether the element has been newly allocated (`false`) or was already present (`true`).

The function `size()` returns the number of allocated elements. You can use the array access operator `[ ]` to iterate over the elements of the container.

#### Note:

The array access operator `[ ]` should only be used in a sequential section of the code. For example, it can be used in the destructor of the PMI.

---

## Debugging Physical Model Interfaces

Print statements represent the simplest approach for debugging a PMI. They can be inserted anywhere in the code to print the values of a variable.

## Chapter 39: Physical Model Interface

Introduction to the Physical Model Interface

For example:

```
void Auger_Recombination::  
compute (const Input& input, Output& output)  
  
{ output.r = C * (input.n + input.p) *  
    (input.n*input.p - input.nie*input.nie);  
if (output.r < 0.0) {  
    output.r = 0.0;  
}  
std::cout << "n = " << input.n << std::endl;  
std::cout << "p = " << input.p << std::endl;  
std::cout << "nie = " << input.nie << std::endl;  
std::cout << "r = " << output.r << std::endl;  
}
```

You also can use a debugger to catch errors in a PMI subroutine. The following instructions apply to `gdb`, the GNU debugger. The same approach also can be adjusted to work with other debuggers:

1. Compile the PMI source code in debug mode by using the `-g` option:

```
cmi -g pmi_Auger.C
```

2. Determine the name of the Sentaurus Device binary. The command:

```
sdevice -@ldd
```

should produce output similar to the following:

```
path to executable: /usr/sentaurus/tcad/T-2022.03/linux64/bin  
executable: sdevice-33.0.1098  
  
/usr/sentaurus/tcad/T-2022.03/linux64/bin/sdevice-33.0.1098:  
ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux),  
dynamically linked (uses shared libs), for GNU/Linux 2.6.18,  
stripped
```

In this example, `/usr/sentaurus/tcad/T-2022.03/linux64/bin/sdevice-33.0.1098` is the name of the Sentaurus Device binary.

3. Start the debugger `gdb` on the Sentaurus Device binary:

```
gdb /usr/sentaurus/tcad/T-2022.03/linux64/bin/sdevice-33.0.1098
```

4. Verify the setting of the environment variables:

```
show environment
```

In particular, the variables `STROOT` and `STRELEASE` must be defined properly. If necessary, use the following `gdb` command to define these variables:

```
set environment STROOT /usr/sentaurus  
set environment STRELEASE T-2022.03
```

## Chapter 39: Physical Model Interface

Introduction to the Physical Model Interface

5. You cannot define a break point in the PMI code until the corresponding shared object file has been loaded. Therefore, run your simulation:

```
run pp1_des.cmd
```

Watch for messages regarding the loading of shared object files. All the shared object files for PMIs are loaded at the very beginning, usually within seconds of starting Sentaurus Device. Afterwards, use the shortcut keys Ctrl+C to interrupt the simulation.

6. Define a break point in the PMI source code. For example:

```
break pmi_Auger.C:100
```

This command inserts a break point on line 100 in the `pmi_Auger.C` file.

7. After defining all required break points, resume the simulation with the command:

```
continue
```

The debugger will now stop whenever the control flow reaches a break point, and all the standard `gdb` commands are available for debugging.

---

## Runtime Support for Vertex-Based PMI Models

Inside vertex-based PMI models, you can access several functions described here. Essentially, they are split into two groups, namely, functions that are valid at the scope of the PMI model and functions that are valid only within the scope of `compute` functions.

### Support at Model Scope

A standard vertex-based PMI is derived from the base class `PMI_Vertex_Interface`; whereas, a simplified vertex-based PMI is derived from the base class `PMI_Vertex_Base`.

Both base classes are derived from the `PMI_Vertex_Common_Base` class and provide the following shared functionality:

```
const char* Name () const;
const char* Filename () const;
const char* ReadRegionName () const;
const char* ReadRegionMaterial () const;
des_materialgroup ReadRegionMaterialGroup () const;

const char* ReadDeviceName () const;

const PMIBaseParam* ReadParameter (const char* name
                                    [, const char* modelName]) const;
double InitParameter (const char* name, double defaultvalue
                      [, const char* modelName]) const;
void InitParameter (const char* name, std::vector<double>& value
                     [, const char* modelName]) const;
```

## Chapter 39: Physical Model Interface

### Introduction to the Physical Model Interface

```
const char* InitStringParameter (const char* name,
                                const char* defaultvalue
                                [, const char* modelName]) const;
void InitStringParameter (const char* name,
                         std::vector<const char*>& value
                         [, const char* modelName]) const;
void double InitModelParameter (const char* name,
                                const char* modelName,
                                double defaultvalue);
void double InitOptoModelParameter(const char* name, double
                                    defaultvalue);

int ReadDimension () const;
void ReadReferenceCoordinates (double ref [3][3]) const;

size_t NumberOfMSConfigStates (const std::string& msconfig_name)
                                const;
const std::string& MSConfigStateName (const std::string&
                                         msconfig_name, size_t
                                         state_index) const;
des_loginfo* LogData () const;
```

The method `Name()` returns the name of the PMI model as specified in the command file. Similarly, `Filename()` returns the name of the corresponding shared object file.

For the current region, call `ReadRegionName()` to determine its name, `ReadRegionMaterial()` to return the name of the region material, and `ReadRegionMaterialGroup()` to find the material group.

The possible values for the material group are:

```
PMI_Vertex_Common_Base::conductor
PMI_Vertex_Common_Base::insulator
PMI_Vertex_Common_Base::semiconductor
PMI_Vertex_Common_Base::unknown
```

`ReadDeviceName()` returns the name of the device.

The methods `ReadParameter()`, `InitParameter()`, and `InitStringParameter()` read the value of a parameter from the parameter file or command file (see [Parameter File of Sentaurus Device on page 1232](#) and [Command File of Sentaurus Device on page 1230](#)). In these methods, `modelName` is an optional argument that allows the value of a parameter to be read from a different PMI model.

The method `InitModelParameter()` allows PMI access to any built-in Sentaurus Device mole-fraction or constant parameters. `modelName` specifies the built-in model and `name` is the parameter within the specified model. When no valid model name or parameter name is found, the provided `defaultvalue` is returned instead. `InitModelParameter` is available only in the `Compute` function not in the PMI constructor. Similarly, the `InitOptoModelParameter()` method gives the PMI access to built-in numeric

## Chapter 39: Physical Model Interface

### Introduction to the Physical Model Interface

optoelectronic parameters. In this case, `name` is the full path to the parameter. `ReadDimension()` returns the dimension of the problem.

#### Note:

The `InitModelParameter()` and `InitOptoModelParameter()` methods only support access to scalar parameters. You cannot access a specific component of a vector entry such as `Physics/Optics/OpticalGeneration/TimeDependence/WaveTime`.

The method `ReadReferenceCoordinates()` provides access to the reference coordinate system. It will return the identity matrix:

$$\text{ref} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (1331)$$

in the case of the unified coordinate system (UCS). Otherwise, another coordinate system matrix is returned.

Multistate configuration-dependent models can call the `NumberOfMSConfigStates()` and `MSConfigStateName()` methods. They return the number of states and the name of a state, respectively.

The function `LogData()` returns the `des_loginfo` class, which can be used to print messages from PMI functions to the log file (see [Runtime Support to Write PMI Output to Log File on page 1259](#)).

## Reaction-Diffusion Species Interface (Model Scope)

The interface to reaction-diffusion (RD) species is the same for both standard and simplified PMI models, as it is provided in the common base class `PMI_Vertex_Common_Base`.

The interface provides the following functions:

- `size_t RDSpecies_size()` returns the number of RD species defined for the actual device. Note that not all of these species need to be defined in the actual device region.
- `void get_RDSpecies_name (size_t isp, std::string& spname)` returns the name of the species with device index `isp`.
- `bool RDSpecies_is_defined (size_t isp)` returns true if the species with device index `isp` is defined in the actual region.

For simplified PMI models, the values for RD species concentrations can be extracted in the `compute` function from the `Input` interface (see [Reaction-Diffusion Species Interface \(Compute Scope\) on page 1243](#)).

**Note:**

Standard PMI models do not have such an interface, that is, accessing RD species concentrations is not supported in this interface. However, within all PMI models, you can read such values using the `ReadScalar(const char*)` function.

## Support at Compute Scope

The following interface functions depend on the local vertex where a model is evaluated. They are not available in the constructor of the PMI, but should only be called in the `Compute` function. In the case of the standard interface, these functions are a part of the base class `PMI_Vertex_Interface`; whereas, the simplified interface provides them through the base class `PMI_Vertex_Input_Base`:

```
void ReadCoordinate (double& x, double& y, double& z) const;
void ReadNearestInterfaceNormal (double& nx, double& ny, double& nz)
    const;
double ReadDistanceFromSemiconductorInsulatorInterface() const;
double ReadDistanceFromSemiSemiInterface() const;
double ReadDistanceFromHighkInsulator() const;
int ReadNearestInterfaceOrientation() const;

double ReadLayerThickness () const;
double ReadLayerThicknessField () const;

double ReadTime () const;
double ReadTransientStepSize () const;
PMI_StepType ReadTransientStepType () const;

double ReadxMoleFraction () const;
double ReadyMoleFraction () const;

double ReadDoping (PMI_DopingSpecies species) const;
double ReadDoping (const char* SpeciesName) const;

double ReadDielectricConstant() const;
double ReadSemiconductorDielectricConstant() const;

int ReadDopingWell () const;

int IsUserFieldDefined (PMI_UserFieldIndex index) const;
double ReadUserField (PMI_UserFieldIndex index) const;
void WriteUserField (PMI_UserFieldIndex index, double value) const;

double ReadStress (PMI_StressIndex index) const;

bool ReadMSCOccupations (const std::string& msc_name, double* values)
    const;

double ReadeSHEDistribution (double energy) const;
double ReadhSHEDistribution (double energy) const;
```

## Chapter 39: Physical Model Interface

### Introduction to the Physical Model Interface

```
double ReadeSHETotalDOS (double energy) const;
double ReadhSHETotalDOS (double energy) const;
double ReadeSHETotalGSV (double energy) const;
double ReadhSHETotalGSV (double energy) const;
```

#### Note:

The support functions for the simplified interface use the data type `pmi_float`; whereas, the support functions for the standard interface use the data type `double`. However, their functionality is identical, and only the `double` version is documented.

The function `ReadCoordinate()` provides the coordinates of the current vertex [ $\mu\text{m}$ ].

The function `ReadNearestInterfaceNormal()` provides the components of a unit vector in the direction of the nearest interface normal.

`ReadDistanceFromSemiconductorInsulatorInterface()` returns the distance of the current vertex to the nearest semiconductor–insulator interface (in  $\mu\text{m}$ ) if the current vertex is in a semiconductor region; otherwise, the function returns minus that distance.

`ReadDistanceFromSemiSemiInterface()` returns the distance of the current vertex to the nearest semiconductor–semiconductor interface (in  $\mu\text{m}$ ). If no semiconductor–semiconductor interface is found in the structure, then the function returns a value  $< 0$ .

`ReadDistanceFromHighkInsulator()` returns the distance of the current vertex to the nearest high-k insulator (in  $\mu\text{m}$ ). If no high-k insulator is found in the structure, then the function returns a value  $< 0$ .

`ReadNearestInterfaceOrientation()` returns the auto-orientation framework orientation (as a three-digit integer) that is closest to the actual orientation at the nearest interface vertex (see [Auto-Orientation Framework on page 86](#)).

`ReadLayerThickness()` returns the value of the `LayerThickness` array [ $\mu\text{m}$ ] for the current vertex (see [Extracting Layer Thickness on page 379](#)).

`ReadLayerThicknessField()` returns the value of the `LayerThicknessField` array [ $\mu\text{m}$ ] for the current vertex (see [Extracting Layer Thickness on page 379](#)).

The functions `ReadTime()` and `ReadTransientStepSize()` return the simulation time and the current step size during a transient simulation [s]. `ReadTransientStepType()` provides access to the actual transient step type.

The enumeration type `PMI_StepType` is defined for identification:

```
enum PMI_StepType {
    PMI_UndefStepType = 0,
    PMI_TR = 1,
    PMI_BDF = 2,
    PMI_BE = 3
}
```

## Chapter 39: Physical Model Interface

### Introduction to the Physical Model Interface

The methods `ReadxMoleFraction()` and `ReadyMoleFraction()` return the x- and y-mole fractions, respectively.

The methods `ReadDoping(species)` and `ReadDoping(SpeciesName)` return the doping profiles for the current vertex [cm<sup>-3</sup>]. The string `SpeciesName` is the same as in the file `datexcodes.txt` (see [Specifying Doping Species on page 60](#)). The enumeration type `PMI_DopingSpecies` is used to select the doping species, the incomplete ionization doping species, and their derivatives:

```
enum PMI_DopingSpecies {
    // Acceptors
    PMI_BoronActive,           // active Boron concentration
    PMI_BoronChemical,         // chemical Boron concentration
    PMI_AluminumActive,        // active Aluminum concentration
    PMI_AluminumChemical,      // chemical Aluminum concentration
    PMI_IndiumActive,          // active Indium concentration
    PMI_IndiumChemical,        // chemical Indium concentration
    PMI_PDopantActive,         // active PDopant concentration
    PMI_PDopantChemical,       // chemical PDopant concentration
    PMI_Acceptor,              // total acceptor concentration

    // incomplete ionization entries
    PMI_AcceptorMinus,         // total incomplete ionization acceptor
                               // concentration
    PMI_AcceptorMinusPer_hDensity,
    PMI_AcceptorMinusPerT,

    // Donors
    PMI_PhosphorusActive,       // active Phosphorus concentration
    PMI_PhosphorusChemical,     // chemical Phosphorus concentration
    PMI_ArsenicActive,          // active Arsenic concentration
    PMI_ArsenicChemical,        // chemical Arsenic concentration
    PMI_AntimonyActive,         // active Antimony concentration
    PMI_AntimonyChemical,       // chemical Antimony concentration
    PMI_NitrogenActive,          // active Nitrogen concentration
    PMI_NitrogenChemical,        // chemical Nitrogen concentration
    PMI_NDopantActive,          // active NDopant concentration
    PMI_NDopantChemical,        // chemical NDopant concentration
    PMI_Donor,                  // total donor concentration

    // incomplete ionization entries
    PMI_DonorPlus,              // total incomplete ionization donor
                               // concentration
    PMI_DonorPlusPer_eDensity,
    PMI_DonorPlusPerT

    // additional species
    PMI_Carbon,
    PMI_CarbonChemical          // chemical Carbon concentration
};
```

## Chapter 39: Physical Model Interface

Introduction to the Physical Model Interface

### Note:

The species `PMI_Acceptor` and `PMI_Donor` are always defined. The remaining entries are only defined if they occur in the simulated device. The incomplete ionization entries are only accessible if the option `IncompleteIonization` is activated (see [Chapter 13 on page 339](#)).

The `ReadDielectricConstant` and `ReadSemiconductorDielectricConstant` methods return the dielectric constant for the current vertex. The second method accounts for only the dielectric constant in semiconductor.

The `ReadDopingWell()` method returns the index of the doping well for the current vertex (see [Initial Guess for Electrostatic Potential and Quasi-Fermi Potentials in Doping Wells on page 234](#)).

The method `IsUserFieldDefined()` checks whether a user-defined field has been specified. The enumeration type `PMI_UserFieldIndex` selects the required field:

```
enum PMI_UserFieldIndex {
    PMI_UserField0, PMI_UserField1, ..., PMI_UserField99
};
```

If a specific user-field is defined, then the method `ReadUserField()` reads its value for the current vertex. For time-dependent user fields, this is the value written in the last successful time step. `WriteUserField()` allows the modification of the value of a specific user-field for the current vertex. It writes to an auxiliary field. After a transient time step is finished, this auxiliary field becomes the field accessible with `ReadUserField()`.

The method `ReadStress()` returns the value of one of the following stress components:

```
enum PMI_StressIndex {
    PMI_StressXX, PMI_StressYY, PMI_StressZZ,
    PMI_StressYZ, PMI_StressXZ, PMI_StressXY
};
```

### Reaction–Diffusion Species Interface (Compute Scope)

For simplified PMI models, the class `PMI_Vertex_Input_Base` provides an interface for the extraction of RD species concentrations. Accessing the species concentrations through this interface supports the automatic derivative computations of the model quantity with respect to the species concentrations. For standard PMI models, such an interface is not supported.

If the interface is enabled for the actual PMI model, then the following function is available:

- `bool RDSpecies_supported()` accesses the actual `Input` class support to RD species and their concentrations. If there is no support, then the other functions cannot be used. Most of the models do not support this interface.

## Chapter 39: Physical Model Interface

Introduction to the Physical Model Interface

If the `RDSpecies_supported()` function returns `true`, then you can use additional functions:

- `size_t RDSpecies_size()` returns the number of RD species defined for the actual device. Note that not all of these species can be defined in the actual region.
- `void get_RDSpecies_name (size_t isp, std::string& spname)` returns the name of the species with device index `isp`.
- `bool RDSpecies_is_defined (size_t isp)` returns true if the species with device index `isp` is defined in the actual region.
- `pmi_float RDSpecies_concentration (size_t isp)` returns the value of the concentration of the RD species with device index `isp`.

## Experimental Functions

Vertex-based PMI models also have access to the following runtime functions:

```
double ReadScalar (const char* name) const;
void ReadVector (const char* name, double vector [3]) const;
```

These functions can be called in `Compute` and provide access to scalar and vector data of Sentaurus Device. See [Table 195 on page 1516](#) and [Table 196 on page 1552](#) for an overview of available data.

### Note:

This is an experimental feature. It is provided as is, without warranty of any kind. In particular, you should be aware of the following limitations:

- Whenever you access additional data in Sentaurus Device, you introduce new model dependencies. However, Sentaurus Device cannot take into account the corresponding derivatives. Therefore, the convergence of the Newton solver might be affected.
- You might introduce cyclic dependencies, which will result in an infinite loop.

---

## Runtime Support for Vertex-Based Multistate Configuration-Dependent Models

The base class `PMI_MSC_Vertex_Interface` provides the same support as `PMI_Vertex_Interface`, plus additional functions needed by models that depend on a multistate configuration (see [Chapter 18 on page 584](#)):

```
class PMI_MSC_Vertex_Interface : public PMI_Vertex_Interface
{
public:
    PMI_MSC_Vertex_Interface(const PMI_Environment&,
                           const std::string& msconfig_name,
```

## Chapter 39: Physical Model Interface

Introduction to the Physical Model Interface

```
int model_index,  
const std::string& model_string);  
  
const std::string& msconfig_name () const;  
size_t nb_states () const;  
std::string& state ( size_t index ) const;  
int model_index () const;  
const std::string& model_string () const;  
virtual void init_parameter (){};  
};
```

### Note:

In the case of the simplified interface, the base class `PMI_MSC_Vertex_Base` is used instead. However, it provides the same functionality as the base class `PMI_MSC_Vertex_Interface` and, therefore, it is not documented separately.

The constructor argument `msconfig_name` determines the name of the multistate configuration on which the model depends, and the constructor arguments `model_index` and `model_string` are an integer and a string that you can evaluate in your model. You call the constructor `PMI_MSC_Vertex_Interface` only indirectly using constructors of base classes of multistate configuration-dependent PMIs.

The function `nb_states` returns the number of states in the selected multistate configuration, `state` returns the name of a particular state, and `msconfig_name`, `model_index`, and `model_string` return the arguments of the constructor of the same name.

The function `init_parameter` is always called before the parameters are changed. It allows you to keep model-internal data up-to-date in cases such as ramping of parameters.

---

## Runtime Support for Mesh-Based PMI Models

A standard mesh-based PMI is derived from the base class `PMI_Device_Interface`; whereas, a simplified mesh-based PMI is derived from the base class `PMI_Device_Base`.

Both base classes provide the following shared functionality:

```
const char* Name () const;  
  
const PMIBaseParam* ReadParameter (const char* name  
                                    [, const char* modelName]) const;  
double InitParameter (const char* name, double defaultvalue  
                      [, const char* modelName]) const;  
void InitParameter (const char* name, std::vector<double>& value  
                      [, const char* modelName]) const;  
const char* InitStringParameter (const char* name,  
                                 const char* defaultvalue  
                                 [, const char* modelName]) const;  
void InitStringParameter (const char* name,
```

## Chapter 39: Physical Model Interface

### Introduction to the Physical Model Interface

```
std::vector<const char*>& value  
[, const char* modelName]) const;  
  
const des_mesh* Mesh () const;  
des_data* Data () const;  
des_loginfo* LogData () const;
```

The method `Name()` returns the name of the PMI model as specified in the command file. The methods `ReadParameter()`, `InitParameter()`, and `InitStringParameter()` read the value of a parameter from the parameter file or command file (see [Parameter File of Sentaurus Device on page 1232](#) and [Command File of Sentaurus Device on page 1230](#)). In these methods, `modelName` is an optional argument that allows the value of a parameter to be read from a different PMI model.

**Note:**

Parameters for a mesh-based PMI must appear in the global parameter section in the parameter file. Regionwise or materialwise parameters are not supported.

The methods `Mesh()` and `Data()` provide access to the mesh and data of Sentaurus Device (see [Device Mesh on page 1247](#) and [Device Data on page 1254](#)).

**Note:**

For the standard interface, the method `Data()` returns a variable of type `des_data`. For the simplified interface, it returns a variable of type `device_data`. The type `des_data` is based on the data type `double`, and `sdevice_data` uses the data type `pmi_float`. Otherwise, their functionality is identical.

The following interface functions should only be called in the `Compute` function, but not in the constructor. The standard interface provides them as members of the base class `PMI_Device_Interface`; whereas, the simplified interface provides them as members of the base class `PMI_Device_Input_Base`:

```
double ReadTime () const;  
double ReadTransientStepSize () const;  
PMI_StepType ReadTransientStepType () const;  
double ReadLayerThickness (int vertex) const;  
double ReadLayerThicknessField (int vertex) const;
```

**Note:**

The support functions for the simplified interface use the data type `pmi_float`; whereas, the support functions for the standard interface use the data type `double`. However, their functionality is identical, and only the `double` version is documented.

The functions `ReadTime()` and `ReadTransientStepSize()` return the simulation time and the current step size during a transient simulation [s]. `ReadTransientStepType()` provides access to the actual transient step type.

## Chapter 39: Physical Model Interface

Introduction to the Physical Model Interface

The enumeration type `PMI_StepType` is defined for identification:

```
enum PMI_StepType {
    PMI_UndefStepType = 0,
    PMI_TR = 1,
    PMI_BDF = 2,
    PMI_BE = 3
}
```

The functions `ReadLayerThickness()` and `ReadLayerThicknessField()` return the corresponding values `LayerThickness` and `LayerThicknessField` (see [Extracting Layer Thickness on page 379](#)).

The function `LogData()` returns the `des_loginfo` class, which can be used to print messages from PMI functions to the log file (see [Runtime Support to Write PMI Output to Log File on page 1259](#)).

## Device Mesh

A device mesh of Sentaurus Device consists of a number of regions. A region is either a contact region consisting of a list of contact vertices or a bulk region consisting of a list of elements. An element is described by a list of vertices.

### Vertex

In the file `PMIModels.h`, the class `des_vertex` is declared as follows:

```
class des_vertex {

public:
    size_t index () const;
    size_t element_vertex_index (const des_element* element) const;

    const double* coord () const;
    double coord (size_t i) const;

    bool equal_coord (des_vertex* v) const;

    size_t size_edge () const;
    des_edge* edge (size_t i) const;

    size_t size_element () const;
    des_element* element (size_t i) const;

    size_t size_region () const;
    des_region* region (size_t i) const;

    size_t size_regioninterface () const;
    des_regioninterface* regioninterface (size_t i) const;
};
```

## Chapter 39: Physical Model Interface

### Introduction to the Physical Model Interface

The value of `index()` can be used as an index for vertex-based data (see [Device Data on page 1254](#)). Similarly, the value of `element_vertex_index()` can be used as an index for element-vertex-based data.

The location of a vertex [ $\mu\text{m}$ ] is given by its coordinates `coord()`. The two versions of the `coord()` function return the coordinates either as a vector or as individual components. You should use the function `equal_coord()` to check whether two vertices have the same coordinates. For example, Sentaurus Device duplicates vertices along heterointerfaces. Consequently, two vertices with different indices can share the same coordinates.

`size_edge()` returns the number of edges connected to a vertex. The method `edge()` can be used to retrieve the  $i$ -th edge.

`size_region()` returns the number of regions containing a vertex. The method `region()` can be used to retrieve the  $i$ -th region.

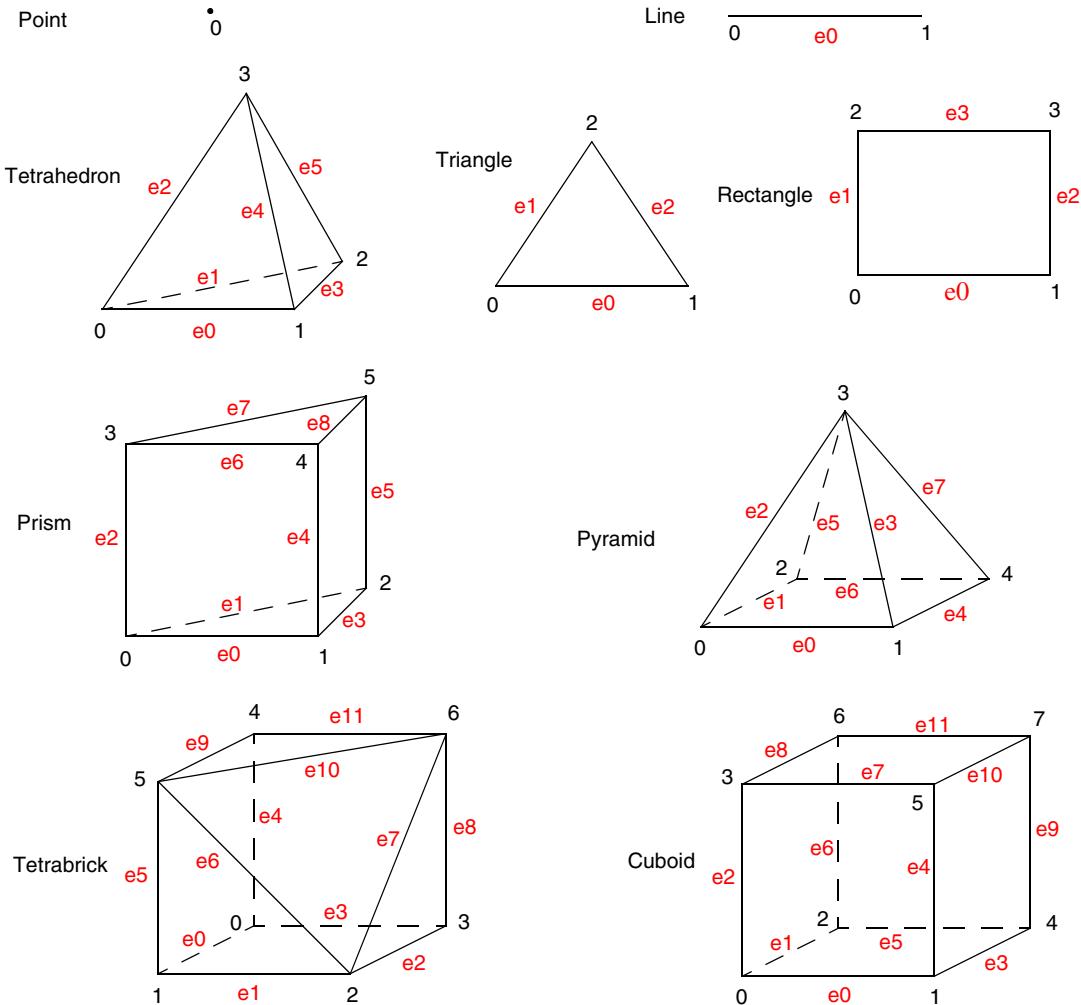
`size_regioninterface()` reports how many region interfaces a vertex belongs to. The  $i$ -th region interface is returned by the method `regioninterface()`.

[Figure 98](#) shows the numbering of vertices and edges for all element types.

## Chapter 39: Physical Model Interface

### Introduction to the Physical Model Interface

Figure 98 Numbering of vertices and edges



### Edge

In the file `PMIModels.h`, the class `des_edge` is declared as follows:

```
class des_edge {
public:
    size_t index () const;
    des_vertex* start () const;
    des_vertex* end () const;
    size_t size_element () const;
    des_element* element (size_t i) const;
```

## Chapter 39: Physical Model Interface

Introduction to the Physical Model Interface

```
    size_t size_region () const;
    des_region* region (size_t i) const;
};
```

You can use the value of `index()` as an index for edge-based data (see [Device Data on page 1254](#)).

`start()` and `end()` return the first and second vertex connected to the edge, respectively.

`size_element()` returns the number of elements connected to an edge. The method `element()` can be used to retrieve the *i*-th element.

`size_region()` returns the number of regions containing an edge. The method `region()` can be used to retrieve the *i*-th region.

### Element

In the file `PMIModels.h`, the class `des_element` is declared as follows:

```
class des_element {

public:
    typedef enum { point, line, triangle, rectangle, tetrahedron,
                  pyramid, prism, cuboid, tetrabrick } des_type;

    size_t index () const;

    des_type type () const;

    size_t size_vertex () const;
    des_vertex* vertex (size_t i) const;

    size_t size_edge () const;
    des_edge* edge (size_t i) const;

    des_bulk* bulk () const;

    size_t element_vertex_offset () const;
};
```

You can use the value of `index()` as an index for element-based data (see [Device Data on page 1254](#)). Similarly, `element_vertex_offset()` returns the start index for element-vertex-based data in this element.

`type()` returns the type of an element (point, line, triangle, rectangle, tetrahedron, pyramid, prism, cuboid, or tetrabrick). An element is mainly described by its vertices. `size_vertex()` returns the number of vertices in an element, and the method `vertex()` can be used to retrieve the *i*-th vertex. `size_edge()` returns the number of edges of an element. The method `edge()` can be used to retrieve the *i*-th edge. The method `bulk()` returns the bulk region containing the element.

## Chapter 39: Physical Model Interface

Introduction to the Physical Model Interface

### Region

In the file `PMIModels.h`, the base class `des_region` is declared as follows:

```
class des_region {  
  
public:  
    typedef enum { bulk, contact } des_type;  
  
    virtual des_type type () const = 0;  
  
    std::string name () const;  
  
    size_t size_vertex () const;  
    des_vertex* vertex (size_t i) const;  
  
    size_t size_edge () const;  
    des_edge* edge (size_t i) const;  
};
```

A mesh of Sentaurus Device consists of two types of region: bulk regions and contacts. The virtual method `type()` returns the type of a region. The name of a region is returned by `name()`. `size_vertex()` returns the number of vertices in a region. The method `vertex()` can be used to retrieve the  $i$ -th vertex. `size_edge()` returns the number of edges in a region. The method `edge()` can be used to retrieve the  $i$ -th edge.

The class `des_bulk` is derived from `des_region`:

```
class des_bulk : public des_region {  
  
public:  
    des_type type () const;  
  
    std::string material () const;  
  
    size_t size_element () const;  
    des_element* element (size_t i) const;  
  
    size_t size_regioninterface () const;  
    des_regioninterface* regioninterface (size_t i) const;  
};
```

`material()` returns the name of the material in a bulk region. `size_element()` returns the number of elements in a region. The method `element()` can be used to retrieve the  $i$ -th element. `size_regioninterface()` reports how many region interfaces are connected to this bulk region. The  $i$ -th region interface can then be retrieved by the method `regioninterface()`.

## Chapter 39: Physical Model Interface

Introduction to the Physical Model Interface

Similarly, the class `des_contact` is also derived from `des_region`:

```
class des_contact : public des_region {  
  
public:  
    des_type type () const;  
};
```

### Region Interface

A region interface separates two bulk regions. It is described by the following class:

```
class des_regioninterface {  
  
public:  
    size_t index () const;  
  
    des_bulk* bulk1 () const;  
    des_bulk* bulk2 () const;  
  
    bool is_heterointerface () const;  
  
    size_t size_vertex () const;  
    des_vertex* vertex (size_t i) const;  
    size_t index (size_t local_vertex_index) const;  
};
```

The value of `index()` is used to access the surface measure array (see [Device Data on page 1254](#)).

The two bulk regions connected to a region interface are returned by `bulk1()` and `bulk2()`.

Use `is_heterointerface()` to determine if double points exist for this interface. For a heterointerface, each vertex belongs to either region 1 or region 2, but not both. For a regular interface, each vertex belongs to both region 1 and region 2.

The number of vertices contained in a region interface is returned by the method `size_vertex()`. The *i*-th vertex can be obtained by invoking `vertex()`. The method `index(size_t local_vertex_index)` is used to obtain the correct index for interface-based data (see [Device Data on page 1254](#)).

### Mesh

In the file `PMIModels.h`, the class `des_mesh` is declared as follows:

```
class des_mesh {  
  
public:  
    int dim () const;  
    void ref_coordinates (double ref [3][3]) const;  
    double ref_coordinates (int i, int j) const;
```

## Chapter 39: Physical Model Interface

### Introduction to the Physical Model Interface

```
size_t size_vertex () const;
size_t size_element_vertex () const;
des_vertex* vertex (size_t i) const;

size_t size_edge () const;
des_edge* edge (size_t i) const;

size_t size_element () const;
des_element* element (size_t i) const;
void find_elements (double x, double y, double z,
                     std::vector<des_element*>& elements) const;

size_t size_region () const;
des_region* region (size_t i) const;

size_t size_regioninterface () const;
des_regioninterface* regioninterface (size_t i) const;
};
```

The dimension of the mesh is given by `dim()`. The possible values are 1, 2, and 3. The two `ref_coordinates()` methods provide access to the reference coordinate system, either to the entire matrix or individual components of the matrix.

They will return the identity matrix:

$$\text{ref} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (1332)$$

in the case of the unified coordinate system. Otherwise, another coordinate system matrix will be returned.

`size_vertex()` returns the number of vertices in the mesh. Similarly, the function `size_element_vertex()` returns the total number of element vertices in the mesh. The method `vertex()` can be used to retrieve the *i*-th vertex. `size_edge()` returns the number of edges in the mesh. The method `edge()` can be used to retrieve the *i*-th edge. `size_element()` returns the number of elements in the mesh. The method `element()` can be used to retrieve the *i*-th element.

For a given point with coordinates (x, y, z) in units of [ $\mu\text{m}$ ], the method `find_elements()` computes a list of elements that contain this point.

`size_region()` returns the number of regions in the mesh. The method `region()` can be used to retrieve the *i*-th region. There are `size_regioninterface()` region interfaces in the mesh, and the *i*-th interface is returned by invoking `regioninterface()`.

## Chapter 39: Physical Model Interface

Introduction to the Physical Model Interface

### Device Data

The class `des_data` provides the following functionality:

```
typedef enum { vertex, edge, element, rivertex, element_vertex }  
    des_location;  
const double*const* ReadCoefficient ();  
const double*const* ReadMeasure ();  
const double*const* ReadSurfaceMeasure ();  
const double*const* ReadContactMeasure ();  
const double* ReadScalar (des_location location, std::string name,  
                           bool update=false);  
const double*const* ReadVector (des_location location,  
                               std::string name,  
                               bool update=false);  
void WriteScalar (des_location location, std::string name,  
                  const double* newvalue);  
const double*const* ReadGradient (des_location location,  
                                 std::string name,  
                                 bool update=false);  
const double* ReadFlux (des_location location, std::string name,  
                        bool update=false);  
size_t NumberOfMSCStates (const std::string& msc_name) const;  
bool ReadMSCStateName (const std::string& msc_name, size_t  
                       state_index, std::string& state_name) const;  
bool ReadMSCOccupations (const std::string& msc_name,  
                           const des_region* region,  
                           double*const* values) const;  
double ReadeSHEDistribution (des_bulk* r, des_vertex* v,  
                             double energy) const;  
double ReadhSHEDistribution (des_bulk* r, des_vertex* v,  
                             double energy) const;  
double ReadeSHEETotalDOS (des_bulk* r, double energy) const;  
double ReadhSHEETotalDOS (des_bulk* r, double energy) const;  
double ReadeSHEETotalGSV (des_bulk* r, double energy) const;  
double ReadhSHEETotalGSV (des_bulk* r, double energy) const;  
double interpolate (des_element* element,  
                    const std::vector<double>& vertex_values,  
                    double x, double y, double z,  
                    des_interpolation interpolation = linear,  
                    double arsinhfactor = 1) const;  
double ReadRTNTrapRate (const std::string& trapName, des_bulk* r,  
                        size_t ce, size_t q, bool& status) const;  
double ReadRTNTrapOccupation (const std::string& trapName, des_bulk* r,  
                            bool& status) const;  
std::vector<std::string> ReadRTNTrapNames (des_bulk* r) const;  
void SetRTNTrapOccupation (const std::string& trapName, des_bulk* r,  
                           double value, bool& status) const;
```

#### Note:

In the case of the simplified interface, the class `sdevice_data` provides the same methods, but uses the data type `pmi_float` instead of `double`. Otherwise, the functionality is identical.

## Chapter 39: Physical Model Interface

### Introduction to the Physical Model Interface

The methods `ReadCoefficient()` and `ReadMeasure()` return the box method coefficients  $\kappa_{ij}$  and measure  $\mu_{ij}$  used in Sentaurus Device (see [Discretization on page 1177](#)).

The `ReadCoefficient()` method returns a two-dimensional array. The two indices are the element index and the local edge number. The units are  $\mu\text{m}^{-1}$  in one dimension, 1 in two dimensions, and  $\mu\text{m}$  in three dimensions.

The following code fragment reads the coefficients for all element edges:

```
const des_mesh* mesh = Mesh();
des_data* data = Data();
const double*const* coeff = data->ReadCoefficient();
for (size_t eli = 0; eli < mesh->size_element(); eli++) {
    des_element* el = mesh->element(eli);
    for (size_t ei = 0; ei < el->size_edge(); ei++) {
        des_edge* e = el->edge(ei);
        const double c = coeff[el->index()][ei];
    }
}
```

#### Note:

The values  $\kappa_{ij}$  returned by `ReadCoefficient()` are element–edge coefficients. The edge coefficients  $\kappa_i$  can be obtained by adding the contributions from all elements connected to an edge  $i$ .

The `ReadMeasure()` method returns a two-dimensional array. The two indices are the element index and the local vertex number. The units are  $\mu\text{m}$  in one dimension,  $\mu\text{m}^2$  in two dimensions, and  $\mu\text{m}^3$  in three dimensions.

The following code fragment reads the measures for all element vertices:

```
const des_mesh* mesh = Mesh();
des_data* data = Data();
const double*const* measure = data->ReadMeasure();
for (size_t eli = 0; eli < mesh->size_element(); eli++) {
    des_element* el = mesh->element(eli);
    for (size_t vi = 0; vi < el->size_vertex(); vi++) {
        des_vertex* v = el->vertex(vi);
        const double m = measure[el->index()][vi];
    }
}
```

#### Note:

The values  $\mu_{ij}$  returned by `ReadMeasure()` are element–vertex measures. The node measures  $\mu_i$  can be obtained by adding the contributions from all elements connected to a vertex  $i$ .

The method `ReadSurfaceMeasure()` provides the surface measure associated with region interface vertices (edge length [ $\mu\text{m}$ ] in two dimensions and surface area [ $\mu\text{m}^2$ ] in three dimensions). The two indices are the region interface index and the local vertex number.

## Chapter 39: Physical Model Interface

### Introduction to the Physical Model Interface

The method `ReadContactMeasure()` provides the contact measure associated with contact vertices (edge length [cm] in two dimensions and contact area [cm<sup>2</sup>] in three dimensions). The two indices are the contact index and the local vertex number.

By default, the PMI *read* functions use a caching mechanism to access internal data entries (for performance reasons). The cache is updated after computing the initial values for the Newton process and after convergence of Newton iterations. This means that, during Newton iterations, the read functions return unchanged arrays. The `ReadScalar()`, `ReadVector()`, `ReadGradient()`, and `ReadFlux()` methods have an additional parameter `update` (default is `false`). If `update=true`, then these methods update array values for each call to that function. The standard call of these methods (`update=false`) is a good solution for postprocessing PMI functions (see [Current Plot File on page 1439](#)). The `update=false` call of these methods is necessary for PMI models that are used during the Newton process (see [Preprocessing for Newton Iterations and Newton Step Control on page 1446](#)).

The methods `ReadScalar()`, `ReadVector()`, and `WriteScalar()` provide access to the data of Sentaurus Device. The values can be located on vertices, edges, elements, or region interfaces. See [Table 195 on page 1516](#) and [Table 196 on page 1552](#) for an overview of available scalar and vector data.

`ReadScalar()` returns a read-only one-dimensional array. Use the `index()` method in the classes `des_vertex`, `des_edge`, or `des_element` to access the array elements. The proper addressing of region interface datasets is shown in the code fragment below.

`ReadVector()` returns a read-only two-dimensional array. The first index selects the dimension (0, 1, 2) and the second index is used in the same way as for scalar data.

`WriteScalar()` writes back a one-dimensional array. The organization of the array is the same as for the arrays obtained with `ReadScalar()`. You must ensure that the size of the array is sufficient to hold all required entries.

`ReadGradient()` returns a 2D array that contains, for each vertex, the gradient of a chosen variable. Thereby, the first index selects the partial derivative:

$$[0] = \frac{\partial}{\partial x}, [1] = \frac{\partial}{\partial y}, [2] = \frac{\partial}{\partial z} \quad (1333)$$

The second index identifies the vertex. The actual implementation of `ReadGradient()` works for vertex-based datasets only.

`ReadFlux()` returns an array that contains, for each vertex, the surface integral of the gradient of a chosen variable taken over the boundary of the box divided by the box volume. The actual implementation of `ReadFlux()` works for vertex-based datasets only.

## Chapter 39: Physical Model Interface

### Introduction to the Physical Model Interface

The following code fragment traverses all region interfaces and reads the surface measure and a dataset for each region interface vertex:

```
const des_mesh* mesh = Mesh();
des_data* data = Data();
const double*const* surface = data->ReadSurfaceMeasure();
const double* hot_elec = data->ReadScalar(des_data::rivertex,
                                             "HotElectronInj");
for (size_t rii = 0; rii < mesh->size_regioninterface(); rii++) {
    des_regioninterface* ri = mesh->regioninterface(rii);
    for (size_t vi = 0; vi < ri->size_vertex(); vi++) {
        des_vertex* v = ri->vertex(vi);
        const double sm = surface[ri->index()][vi];
        const double he = hot_elec[ri->index(vi)];
    }
}
```

The following code fragment traverses all contacts and reads the contact measure for each contact vertex:

```
const des_mesh* mesh = Mesh();
des_data* data = Data();
const double*const* contact_measure = data->ReadContactMeasure();
for (size_t ri = 0; ri < mesh->size_region(); ri++) {
    des_region* r = mesh->region(ri);
    if (r->type() == des_region::contact) {
        for (size_t vi = 0; vi < r->size_vertex(); vi++) {
            const double m = contact_measure[ri][vi];
        }
    }
}
```

The method `interpolate()` can be called to interpolate a vertex-based field within an element. The location of the interpolation is given by the coordinates (x, y, z) in units of [ $\mu\text{m}$ ]. The field values  $v_i$  on the element vertices must be supplied in the vector `vertex_values`.

The following code fragment shows the proper usage:

```
std::vector<double> vertex_values;
double x = 1.0; // interpolation location
double y = 1.0;
double z = 1.0;

for (size_t vi = 0; vi < element->size_vertex (); vi++) {
    des_vertex* v = element->vertex (vi);
    double vertex_value = 1.0; // value of field on vertex v
    vertex_values.push_back (vertex_value);
}
double result = data->interpolate (element, vertex_values, x, y, z,
                                    des_data::arsinh, 10.0);
```

## Chapter 39: Physical Model Interface

### Introduction to the Physical Model Interface

The following interpolation modes are supported:

```
des_data::linear  
des_data::logarithmic  
des_data::arsinh
```

Depending on the interpolation mode, the interpolated value  $v$  is given by:

- linear:  $v = \sum_i w_i v_i$
- logarithmic:  $v = e^{\sum_i w_i \log v_i}$
- arsinh:  $v = \frac{1}{f} \sinh \sum_i w_i \operatorname{asinh}(fv_i)$

Sentaurus Device determines the weights  $w_i$  based on the distance of the point  $(x, y, z)$  from the vertices of the element. The weights are nonnegative and add up to 1.

The methods `ReadRTNTrapNames()`, `ReadRTNTrapRate()`, and `ReadRTNTrapOccupation()` are access functions to read the trap names, trap capture and emission rates, and trap occupation of random telegraph noise (RTN) traps. The method `SetRTNTrapOccupation()` can be used to set the occupation of RTN traps.

The following code shows the proper usage:

```
des_region* r = Mesh()->region(0);  
des_bulk* b = dynamic_cast<des_bulk*>(r);  
// read RTN trap names  
std::vector<std::string> trapNames = Data()->ReadRTNTrapNames(b);  
for ( auto nn:trapNames ) {  
    // read RTN trap occupation  
    bool status = false;  
    pmi_float trapOccupation =  
        Data()->ReadRTNTrapOccupation(nn,b,status);  
    // read electron emission rate to conduction band  
    pmi_float rec = Data()->ReadRTNTrapRate(nn,b,1,0,status);  
    // read electron emission rate to valence band  
    pmi_float rev = Data()->ReadRTNTrapRate(nn,b,1,1,status);  
    // read electron capture rate from conduction band  
    pmi_float rcc = Data()->ReadRTNTrapRate(nn,b,0,0,status);  
    // read electron capture rate from valence band  
    pmi_float rcv = Data()->ReadRTNTrapRate(nn,b,0,1,status);  
    // set RTN trap occupation  
    Data()->SetRTNTrapOccupation(nn,b,1.0,status);  
}
```

## Runtime Support to Write PMI Output to Log File

The `PMIModels.h` file contains the `des_loginfo` class, which can be used to print messages from PMI functions to the log file:

```
class des_loginfo {  
  
    FILE* GetLogFileToWriteOutput();  
  
    void PMIPrint(const char *format, ...);  
}
```

The method `GetLogFileToWriteOutput()` returns a pointer to the log file, and the method `PMIPrint()` writes the message to the log file. The `PMIPrint()` method internally calls the `GetLogFileToWriteOutput()` method to retrieve log file information. This class can be accessed from both vertex-based and mesh-based PMIs.

The following code fragment uses the `des_loginfo` class to print an integer to the log file:

```
des_loginfo* LogInfo = LogData();  
  
LogInfo->PMIPrint(" This writes an integer value to Log File %d \n",  
                   integer);
```

---

## Generation–Recombination

This section presents the following PMIs:

- [Avalanche Generation](#)
  - [Generation–Recombination](#)
  - [Lifetimes](#)
  - [Nonlocal Generation–Recombination](#)
  - [Tunneling Parameters](#)
- 

## Avalanche Generation

The generation rate due to impact ionization can be expressed as:

$$G^{\parallel} = \alpha_n n v_n + \alpha_p p v_p \quad (1334)$$

where  $\alpha_n$  and  $\alpha_p$  are the ionization coefficients for electrons and holes, respectively (compare with [Equation 435 on page 487](#)). The PMI allows you to redefine the calculation of  $\alpha_n$  and  $\alpha_p$ .

## Chapter 39: Physical Model Interface

### Generation–Recombination

## Dependencies

The ionization coefficients  $\alpha_n$  and  $\alpha_p$  can depend on the following variables:

F	Driving force [Vcm $^{-1}$ ]
t	Lattice temperature [K]
bg	Band gap [eV]
ct	Carrier temperature [K]
currentWoMob[3]	Current without mobility [cm $^{-4}$ AVs]

The parameter `ct` represents the electron temperature during the calculation of  $\alpha_n$  and the hole temperature during the calculation of  $\alpha_p$ .

The parameter `currentWoMob` can be used to compute anisotropic avalanche generation. Only the first  $d$  components of the vector `currentWoMob` are defined, where  $d$  is equal to the dimension of the problem. It is recommended that only the direction of the vector `currentWoMob` is taken into account, but not its magnitude.

The PMI model must compute the following results:

alpha	Ionization coefficient [cm $^{-1}$ ]
-------	--------------------------------------

In the case of the standard interface, the following derivatives must be computed as well:

dalphadF	Derivative of alpha with respect to F [V $^{-1}$ ]
dalphadt	Derivative of alpha with respect to t [cm $^{-1}$ K $^{-1}$ ]
dalphadbg	Derivative of alpha with respect to bg [cm $^{-1}$ eV $^{-1}$ ]
dalphadct	Derivative of alpha with respect to ct [cm $^{-1}$ K $^{-1}$ ]
dalphadcurrentWoMob[3]	Derivative of alpha with respect to currentWoMob [cm $^3$ A $^{-1}$ V $^{-1}$ s $^{-1}$ ]

Only the first  $d$  components of the vector `dalphadcurrentWoMob` need to be computed.

## Chapter 39: Physical Model Interface

Generation–Recombination

### Standard C++ Interface

Different driving forces for avalanche generation can be selected in the command file. The enumeration type `PMI_AvalancheDrivingForce`, defined in `PMIModels.h`, is used to reflect your selection:

```
enum PMI_AvalancheDrivingForce {
    PMI_AvalancheElectricField,
    PMI_AvalancheParallelElectricField,
    PMI_AvalancheGradQuasiFermi,
    PMI_AvalancheCarrierTemperatureCanali
};
```

The following base class is declared in the file `PMIModels.h`:

```
class PMI_Avalanche : public PMI_Vertex_Interface {

private:
    const PMI_AvalancheDrivingForce drivingForce;

public:
    PMI_Avalanche (const PMI_Environment& env,
                    const PMI_AvalancheDrivingForce force);
    virtual ~PMI_Avalanche ();

    PMI_AvalancheDrivingForce AvalancheDrivingForce () const
        { return drivingForce; }

    virtual void Compute_alpha
        (const double F, const double t, const double bg,
         const double ct, const double currentWoMob[3], double& alpha)
        = 0;

    virtual void Compute_dalphadF
        (const double F, const double t, const double bg,
         const double ct, const double currentWoMob[3], double& dalphadF)
        = 0;

    virtual void Compute_dalphadt
        (const double F, const double t, const double bg,
         const double ct, const double currentWoMob[3], double& dalphadt)
        = 0;

    virtual void Compute_dalphadbg
        (const double F, const double t, const double bg,
         const double ct, const double currentWoMob[3], double& dalphadbg)
        = 0;

    virtual void Compute_dalphadct
        (const double F, const double t, const double bg,
         const double ct, const double currentWoMob[3], double& dalphadct)
        = 0;
```

## Chapter 39: Physical Model Interface

### Generation–Recombination

```
virtual void Compute_dalphadcurrentWoMob
    (const double F, const double t, const double bg,
     const double ct, const double currentWoMob[3],
     double dalphadcurrentWoMob[3]) = 0;
};
```

Two virtual constructors are required for the calculation of the ionization coefficients  $\alpha_n$  and  $\alpha_p$ :

```
typedef PMI_Avalanche* new_PMI_Avalanche_func
    (const PMI_Environment& env, const PMI_AvalancheDrivingForce force);
extern "C" new_PMI_Avalanche_func new_PMI_e_Avalanche;
extern "C" new_PMI_Avalanche_func new_PMI_h_Avalanche;
```

## Simplified C++ Interface

The following base class is declared in the file `PMI.h`:

```
class PMI_Avalanche_Base : public PMI_Vertex_Base {

public:
    class Input : public PMI_Vertex_Input_Base {
public:
    pmi_float F;           // driving force
    pmi_float t;           // lattice temperature
    pmi_float bg;          // band gap
    pmi_float ct;          // carrier temperature
    pmi_float currentWoMob[3]; // current density (without mobility)
};

    class Output {
public:
    pmi_float alpha;        // ionization coefficient
};

    PMI_Avalanche_Base (const PMI_Environment& env,
                        const PMI_AvalancheDrivingForce force);
    virtual ~PMI_Avalanche_Base ();

    PMI_AvalancheDrivingForce AvalancheDrivingForce () const;

    virtual void compute (const Input& input, Output& output) = 0;
};
```

The prototype for the virtual constructor is given as:

```
typedef PMI_Avalanche_Base* new_PMI_Avalanche_Base_func
    (const PMI_Environment& env, const PMI_AvalancheDrivingForce
     force);
extern "C" new_PMI_Avalanche_Base_func new_PMI_e_Avalanche_Base;
extern "C" new_PMI_Avalanche_Base_func new_PMI_h_Avalanche_Base;
```

## Chapter 39: Physical Model Interface

### Generation–Recombination

## Example: Okuto Avalanche Generation

Okuto and Crowell propose the following expression for the ionization coefficient  $\alpha$ :

$$\alpha(F) = a[1 + c(T - T_0)]F \exp\left[-\frac{b[1 + d(T - T_0)]}{F}\right]^2 \quad (1335)$$

This built-in model is discussed in [Okuto–Crowell Model on page 499](#) and its implementation as a PMI model is:

```
#include "PMIModels.h"

class Okuto_Avalanche : public PMI_Avalanche {

protected:
    const double T0;
    double a, b, c, d;

public:
    Okuto_Avalanche (const PMI_Environment& env,
                      const PMI_AvalancheDrivingForce force);

    ~Okuto_Avalanche ();

    void Compute_alpha
        (const double F, const double t, const double bg,
         const double ct, const double currentWoMob[3], double& alpha);

    void Compute_dalphadF
        (const double F, const double t, const double bg,
         const double ct, const double currentWoMob[3], double& dalphadF);

    void Compute_dalphadt
        (const double F, const double t, const double bg,
         const double ct, const double currentWoMob[3], double& dalphadt);

    void Compute_dalphadbg
        (const double F, const double t, const double bg,
         const double ct, const double currentWoMob[3], double& dalphadbg);

    void Compute_dalphadct
        (const double F, const double t, const double bg,
         const double ct, const double currentWoMob[3], double& dalphadct);

    void Compute_dalphadcurrentWoMob
        (const double F, const double t, const double bg,
         const double ct, const double currentWoMob[3], double
          dalphadcurrentWoMob[3]);
};

}
```

## Chapter 39: Physical Model Interface

### Generation–Recombination

```
Okuto_Avalanche::  
Okuto_Avalanche (const PMI_Environment& env,  
                  const PMI_AvalancheDrivingForce force) :  
    PMI_Avalanche (env, force),  
    T0 (300.0)  
{  
}  
  
Okuto_Avalanche::  
~Okuto_Avalanche ()  
{  
}  
  
void Okuto_Avalanche::  
Compute_alpha (const double F, const double t, const double bg,  
              const double ct, const double currentWoMob[3], double&  
              alpha)  
{ const double aa = a * (1.0 + c * (t - T0));  
  const double bb = b * (1.0 + d * (t - T0)) / F;  
  alpha = aa * F * exp (-bb*bb);  
}  
  
void Okuto_Avalanche::  
Compute_dalphadF (const double F, const double t, const double bg,  
                  const double ct, const double currentWoMob[3],  
                  double& dalphadF)  
{ const double aa = a * (1.0 + c * (t - T0));  
  const double bb = b * (1.0 + d * (t - T0)) / F;  
  const double alpha = aa * F * exp (-bb*bb);  
  dalphadF = (alpha / F) * (1.0 + 2.0*bb*bb);  
}  
  
void Okuto_Avalanche::  
Compute_dalphadt (const double F, const double t, const double bg,  
                  const double ct, const double currentWoMob[3],  
                  double& dalphadt)  
{ const double aa = a * (1.0 + c * (t - T0));  
  const double bb = b * (1.0 + d * (t - T0)) / F;  
  const double tmp = F * exp (-bb*bb);  
  dalphadt = tmp * (a * c - 2.0 * aa * bb * b * d / F);  
}  
  
void Okuto_Avalanche::  
Compute_dalphadbg (const double F, const double t, const double bg,  
                  const double ct, const double currentWoMob[3],  
                  double& dalphadbg)  
{ dalphadbg = 0.0;  
}  
  
void Okuto_Avalanche::  
Compute_dalphadct (const double F, const double t, const double bg,  
                  const double ct, const double currentWoMob[3],  
                  double& dalphadct)  
{ dalphadct = 0.0;  
}  
  
void Okuto_Avalanche::
```

## Chapter 39: Physical Model Interface

### Generation–Recombination

```
Compute_dalphadcurrentWoMob (const double F, const double t,
                               const double bg, const double ct,
                               const double currentWoMob[3],
                               double dalphadcurrentWoMob[3])
{ const int dim = ReadDimension ();
  for (int k = 0; k < dim; k++) {
    dalphadcurrentWoMob [k] = 0.0;
  }
}

class Okuto_e_Avalanche : public Okuto_Avalanche {

public:
  Okuto_e_Avalanche (const PMI_Environment& env,
                      const PMI_AvalancheDrivingForce force);

  ~Okuto_e_Avalanche () {}

Okuto_e_Avalanche:::
Okuto_e_Avalanche (const PMI_Environment& env,
                    const PMI_AvalancheDrivingForce force) :
  Okuto_Avalanche (env, force)
{ // default values
  a = InitParameter ("a_e", 0.426);
  b = InitParameter ("b_e", 4.81e5);
  c = InitParameter ("c_e", 3.05e-4);
  d = InitParameter ("d_e", 6.86e-4);
}

class Okuto_h_Avalanche : public Okuto_Avalanche {

public:
  Okuto_h_Avalanche (const PMI_Environment& env,
                      const PMI_AvalancheDrivingForce force);

  ~Okuto_h_Avalanche () {}

Okuto_h_Avalanche:::
Okuto_h_Avalanche (const PMI_Environment& env,
                    const PMI_AvalancheDrivingForce force) :
  Okuto_Avalanche (env, force)
{ // default values
  a = InitParameter ("a_h", 0.243);
  b = InitParameter ("b_h", 6.53e5);
  c = InitParameter ("c_h", 5.35e-4);
  d = InitParameter ("d_h", 5.67e-4);
}

extern "C"
PMI_Avalanche* new_PMI_e_Avalanche
  (const PMI_Environment& env, const PMI_AvalancheDrivingForce force);
{ return new Okuto_e_Avalanche (env, force);
}
```

## Chapter 39: Physical Model Interface

### Generation–Recombination

```
extern "C"
PMI_Avalanche* new_PMI_h_Avalanche
    (const PMI_Environment& env, const PMI_AvalancheDrivingForce force)
{ return new Okuto_h_Avalanche (env, force);
}
```

---

## Generation–Recombination

The recombination rate  $R_{\text{net}}$  appears in the electron and hole continuity equations (see [Equation 57 on page 238](#)).

### Dependencies

The recombination rate  $R_{\text{net}}$  can depend on these variables:

t	Lattice temperature [K]
n	Electron density [ $\text{cm}^{-3}$ ]
p	Hole density [ $\text{cm}^{-3}$ ]
nie	Effective intrinsic density [ $\text{cm}^{-3}$ ]
f	Absolute value of electric field [ $\text{V}\text{cm}^{-1}$ ]

The PMI model must compute the following results:

r	Generation–recombination rate [ $\text{cm}^{-3}\text{s}^{-1}$ ]
---	---

In the case of the standard interface, the following derivatives must be computed as well:

drdt	Derivative of r with respect to t [ $\text{cm}^{-3}\text{s}^{-1}\text{K}^{-1}$ ]
drdn	Derivative of r with respect to n [ $\text{s}^{-1}$ ]
drdp	Derivative of r with respect to p [ $\text{s}^{-1}$ ]
drdnie	Derivative of r with respect to nie [ $\text{s}^{-1}$ ]
drdf	Derivative of r with respect to f [ $\text{cm}^{-2}\text{s}^{-1}\text{V}^{-1}$ ]

## Chapter 39: Physical Model Interface

Generation–Recombination

### Standard C++ Interface

The following base class is declared in the file `PMIModels.h`:

```
class PMI_Recombination : public PMI_Vertex_Interface {

public:
    PMI_Recombination (const PMI_Environment& env);
    virtual ~PMI_Recombination ();
    virtual useCorrectedDensities() {return false;}

    virtual void Compute_r
        (const double t, const double n, const double p,
         const double nie, const double f, double& r) = 0;

    virtual void Compute_drdt
        (const double t, const double n, const double p,
         const double nie, const double f, double& drdt) = 0;

    virtual void Compute_drdn
        (const double t, const double n, const double p,
         const double nie, const double f, double& drdn) = 0;

    virtual void Compute_drdp
        (const double t, const double n, const double p,
         const double nie, const double f, double& drdp) = 0;

    virtual void Compute_drdnie
        (const double t, const double n, const double p,
         const double nie, const double f, double& drdnie) = 0;

    virtual void Compute_drdf
        (const double t, const double n, const double p,
         const double nie, const double f, double& drdf) = 0;
};
```

The prototype for the virtual constructor is:

```
typedef PMI_Recombination* new_PMI_Recombination_func
    (const PMI_Environment& env);
extern "C" new_PMI_Recombination_func new_PMI_Recombination;
```

By default, Sentaurus Device assumes that a PMI generation–recombination model depends on the electric field. However, you can implement the optional function `PMI_Recombination_ElectricField()` to indicate whether the model depends on the electric field.

By default, the generation–recombination PMI passes uncorrected carrier and intrinsic densities regardless of quantum corrections or Fermi statistics activation in the Sentaurus Device command file.

To force Sentaurus Device to pass the corrected carrier and intrinsic densities to the generation–recombination PMI, the virtual function `useCorrectedDensity()` in the PMI must return `true`.

## Chapter 39: Physical Model Interface

### Generation–Recombination

For visualization purposes, two data entries are available:

- `QCEffectiveIntrinsicDensity` is an improved version of `EffectiveIntrinsicDensity` and contains corrections due to Fermi statistics and quantization effects.
- `QCEffectiveBandgap` is the `EffectiveBandgap` with extra narrowing due to the quantum potentials and all the effects implemented through the quantum-potential framework.

If the model does not depend on the electric field (return value of 0), then the method `Compute_drdf()` is not called, and the matrix assembly in Sentaurus Device works more efficiently:

```
typedef int PMI_Recombination_ElectricField_func ();
extern "C"
PMI_Recombination_ElectricField_func PMI_Recombination_ElectricField;
```

## Simplified C++ Interface

The following base class is declared in the file `PMI.h`:

```
class PMI_Recombination_Base : public PMI_Vertex_Base {

public:
    class Input : public PMI_Vertex_Input_Base {
public:
    pmi_float t;      // lattice temperature
    pmi_float n;      // electron density
    pmi_float p;      // hole density
    pmi_float nie;   // effective intrinsic density
    pmi_float f;      // absolute value of electric field
};

    class Output {
public:
    pmi_float r;      // recombination rate
};

    PMI_Recombination_Base (const PMI_Environment& env);
    virtual ~PMI_Recombination_Base ();
    virtual useCorrectedDensities() {return false;}

    virtual void compute (const Input& input, Output& output) = 0;
};
```

The prototype for the virtual constructor is given as:

```
typedef PMI_Recombination_Base* new_PMI_Recombination_Base_func
(const PMI_Environment& env);
extern "C" new_PMI_Recombination_Base_func new_PMI_Recombination_Base;
```

## Chapter 39: Physical Model Interface

### Generation–Recombination

The optional function `PMI_Recombination_ElectricField()` is recognized as well, as in the case of the standard interface.

## Example: Auger Recombination

See [Standard C++ Interface on page 1211](#) and [Simplified C++ Interface on page 1214](#).

---

## Lifetimes

This PMI provides access to the electron and hole lifetimes,  $\tau_n$  and  $\tau_p$ , in the SRH recombination (see [Equation 283 on page 401](#)) and the coupled defect level (CDL) recombination (see [Equation 411 on page 479](#)).

In the command file, the names of the lifetime models are given as arguments to the `SRH` or `CDL` keywords:

```
Physics { Recombination (SRH (pmi_model_name)) }
```

or:

```
Physics { Recombination (CDL (pmi_model_name)) }
```

### Note:

A PMI model overrides all other keywords in an `SRH` or a `CDL` statement.

## Dependencies

A PMI lifetime model can depend on the variable:

`t` Lattice temperature [K]

It must compute the following results:

`tau` Lifetime  $\tau$  [s]

In the case of the standard interface, the following derivative must be computed as well:

`dtaudt` Derivative of  $\tau$  with respect to lattice temperature [ $sK^{-1}$ ]

## Chapter 39: Physical Model Interface

Generation–Recombination

### Standard C++ Interface

The enumeration type `PMI_LifetimeModel` describes where the PMI lifetime is used:

```
enum PMI_LifetimeModel {
    PMI_SRH,
    PMI_CDLL,
    PMI_CDL2
};
```

The following base class is declared in the file `PMIModels.h`:

```
class PMI_Lifetime : public PMI_Vertex_Interface {

private:
    const PMI_LifetimeModel lifetimeModel;

public:
    PMI_Lifetime (const PMI_Environment& env,
                  const PMI_LifetimeModel model);

    virtual ~PMI_Lifetime ();

    PMI_LifetimeModel LifetimeModel () const { return lifetimeModel; }

    virtual void Compute_tau
        (const double t, double& tau) = 0;

    virtual void Compute_dtaudt
        (const double t, double& dtaudt) = 0;
};
```

Two virtual constructors must be implemented for electron and hole lifetimes:

```
typedef PMI_Lifetime* new_PMI_Lifetime_func
    (const PMI_Environment& env, const PMI_LifetimeModel model);
extern "C" new_PMI_Lifetime_func new_PMI_e_Lifetime;
extern "C" new_PMI_Lifetime_func new_PMI_h_Lifetime;
```

### Simplified C++ Interface

The following base class is declared in the file `PMI.h`:

```
class PMI_Lifetime_Base : public PMI_Vertex_Base {

public:
    class Input : public PMI_Vertex_Input_Base {
    public:
        pmi_float t; // lattice temperature
    };

    class Output {
    public:
```

## Chapter 39: Physical Model Interface

### Generation–Recombination

```
    pmi_float tau; // lifetime
};

PMI_Lifetime_Base (const PMI_Environment& env,
                    const PMI_LifetimeModel model);
virtual ~PMI_Lifetime_Base ();

PMI_LifetimeModel LifetimeModel () const;

virtual void compute (const Input& input, Output& output) = 0;
};
```

The prototype for the virtual constructor is given as:

```
typedef PMI_Lifetime_Base* new_PMI_Lifetime_Base_func
(const PMI_Environment& env, const PMI_LifetimeModel model);
extern "C" new_PMI_Lifetime_Base_func new_PMI_e_Lifetime_Base;
extern "C" new_PMI_Lifetime_Base_func new_PMI_h_Lifetime_Base;
```

## Example: Doping- and Temperature-Dependent Lifetimes

The following example combines doping-dependent lifetimes (Scharfetter) and temperature dependence (power law):

$$\tau = \tau_{\min} + \frac{\tau_{\max} - \tau_{\min}}{1 + \frac{N_{A,0} + N_{D,0}}{N_{ref}}} \gamma \frac{T}{300K}^{\alpha} \quad (1336)$$

```
#include "PMIModels.h"

class Scharfetter_Lifetime : public PMI_Lifetime {

protected:
    const double T0;
    double taumin, taumax, Nref, gamma, Talpha;

public:
    Scharfetter_Lifetime (const PMI_Environment& env,
                          const PMI_LifetimeModel model);

    ~Scharfetter_Lifetime ();
    void Compute_tau
        (const double t, double& tau);

    void Compute_dtaudt
        (const double t, double& dtaudt);

};
```

## Chapter 39: Physical Model Interface

### Generation–Recombination

```
Scharfetter_Lifetime::  
Scharfetter_Lifetime (const PMI_Environment& env,  
                      const PMI_LifetimeModel model) :  
    PMI_Lifetime (env, model),  
    T0 (300.0)  
{  
}  
  
Scharfetter_Lifetime::  
~Scharfetter_Lifetime ()  
{  
}  
  
void Scharfetter_Lifetime::  
Compute_tau (const double t, double& tau)  
{ const double Ni = ReadDoping (PMI_Acceptor) + ReadDoping (PMI_Donor);  
    tau = taumin + (taumax - taumin) / (1.0 + pow (Ni/Nref, gamma));  
    tau *= pow (t/T0, Talpha);  
}  
  
void Scharfetter_Lifetime::  
Compute_dtaudt (const double t, double& dtaudt)  
{ const double Ni = ReadDoping (PMI_Acceptor) + ReadDoping (PMI_Donor);  
    dtaudt = taumin + (taumax - taumin) / (1.0 + pow (Ni/Nref, gamma));  
    dtaudt *= (Talpha/T0) * pow (t/T0, Talpha-1.0);  
}  
  
class Scharfetter_e_Lifetime : public Scharfetter_Lifetime {  
  
public:  
    Scharfetter_e_Lifetime (const PMI_Environment& env,  
                           const PMI_LifetimeModel model);  
  
    ~Scharfetter_e_Lifetime () {}  
};  
  
Scharfetter_e_Lifetime::  
Scharfetter_e_Lifetime (const PMI_Environment& env,  
                      const PMI_LifetimeModel model) :  
    Scharfetter_Lifetime (env, model)  
{ taumin = InitParameter ("taumin_e", 0.0);  
    taumax = InitParameter ("taumax_e", 1.0e-5);  
    Nref = InitParameter ("Nref_e", 1.0e16);  
    gamma = InitParameter ("gamma_e", 1.0);  
    Talpha = InitParameter ("Talpha_e", -1.5);  
}  
  
class Scharfetter_h_Lifetime : public Scharfetter_Lifetime {  
  
public:  
    Scharfetter_h_Lifetime (const PMI_Environment& env,  
                           const PMI_LifetimeModel model);  
  
    ~Scharfetter_h_Lifetime () {}  
};
```

## Chapter 39: Physical Model Interface

### Generation–Recombination

```
Scharfetter_h_Lifetime::  
Scharfetter_h_Lifetime (const PMI_Environment& env,  
                      const PMI_LifetimeModel model) :  
    Scharfetter_Lifetime (env, model)  
  
{ taumin = InitParameter ("taumin_h", 0.0);  
  taumax = InitParameter ("taumax_h", 3.0e-6);  
  Nref = InitParameter ("Nref_h", 1.0e16);  
  gamma = InitParameter ("gamma_h", 1.0);  
  Talpha = InitParameter ("Talpha_h", -1.5);  
}  
  
extern "C"  
PMI_Lifetime* new_PMI_e_Lifetime  
  (const PMI_Environment& env, const PMI_LifetimeModel model)  
{ return new Scharfetter_e_Lifetime (env, model);  
}  
  
extern "C"  
PMI_Lifetime* new_PMI_h_Lifetime  
  (const PMI_Environment& env, const PMI_LifetimeModel model)  
{ return new Scharfetter_h_Lifetime (env, model);  
}
```

---

## Nonlocal Generation–Recombination

The nonlocal generation–recombination model computes individual electron and hole recombination rates in [Equation 57 on page 238](#). The name of the PMI model must appear as a recombination model within the `Physics` section of the command file:

```
Physics { Recombination (pmi_model_name) }
```

The computed electron and hole recombination rates also can be plotted in the `Plot` section:

```
Plot {  
    PMIeNonLocalRecombination  
    PMIhNonLocalRecombination  
}
```

## Dependencies

[Nonlocal Interface on page 1218](#) discusses the supported dependencies of nonlocal models. The actual dependencies must be defined with the method `DefineDependencies()`, and the Jacobian matrices must be defined with the method `DefineJacobians()`.

## Chapter 39: Physical Model Interface

### Generation–Recombination

The method `Compute_parallel()` must compute the following results for the vertices defined in the input argument:

<code>elec</code>	Vector of electron recombination rates [ $\text{cm}^{-3}\text{s}^{-1}$ ]
<code>hole</code>	Vector of hole recombination rates [ $\text{cm}^{-3}\text{s}^{-1}$ ]
<code>J_elec</code>	Map of electron Jacobian matrices
<code>J_hole</code>	Map of hole Jacobian matrices

## Standard C++ Interface

The following base class is declared in the file `PMIModels.h`:

```
class PMI_NonLocal_Recombination : public PMI_Device_Interface {

public:
    class Input {
public:
    const des_region* region;           // all vertices belong
                                         // to this region
    const std::vector<int>& vertices;   // list of vertices
};

    class Output {
public:
    std::vector<double>& elec;          // nonlocal recombination
                                         // rates (electrons)
    std::vector<double>& hole;           // nonlocal recombination
                                         // rates (holes)
    des_id_to_jacobian_map& J_elec;     // derivatives (electrons)
    des_id_to_jacobian_map& J_hole;      // derivatives (holes)
};

PMI_NonLocal_Recombination (const PMI_Device_Environment& env);
virtual ~PMI_NonLocal_Recombination ();

virtual void DefineDependencies
(std::vector<des_data::des_id>& dependencies) = 0;

virtual void DefineJacobians (des_id_to_jacobian_map& J_elec,
                             des_id_to_jacobian_map& J_hole) = 0;

virtual void Compute_parallel (const Input& input, Output& output)
= 0;

    virtual bool NeedNewEdges () { return false; }
};
```

## Chapter 39: Physical Model Interface

Generation–Recombination

The prototype for the virtual constructor is:

```
typedef PMI_NonLocal_Recombination*
    new_PMI_NonLocal_Recombination_func
    (const PMI_Device_Environment& env);
extern "C" new_PMI_NonLocal_Recombination_func
    new_PMI_NonLocal_Recombination;
```

## Simplified C++ Interface

The following base class is declared in the file `PMI.h`:

```
class PMI_NonLocal_Recombination_Base : public PMI_Device_Base {

public:
    class Input : public PMI_Device_Input_Base {
public:
    const des_region* region;           // all vertices belong
                                         // to this region
    const std::vector<int>& vertices;   // list of vertices
};

    class Output {
public:
    std::vector<pmi_float>& elec;        // nonlocal recombination
                                         // rates (electrons)
    std::vector<pmi_float>& hole;        // nonlocal recombination
                                         // rates (holes)
    sdevice_id_to_jacobian_map& J_elec;  // derivatives (electrons)
    sdevice_id_to_jacobian_map& J_hole;  // derivatives (holes)
};

    PMI_NonLocal_Recombination_Base (const PMI_Device_Environment&
                                         env);
    virtual ~PMI_NonLocal_Recombination_Base () ;

    virtual void DefineDependencies
        (std::vector<sdevice_data::sdevice_id>& dependencies) = 0;

    virtual void DefineJacobians (sdevice_id_to_jacobian_map& J_elec,
                                 sdevice_id_to_jacobian_map& J_hole)
        = 0;

    virtual void Compute_parallel (const Input& input, Output& output)
        = 0;

    virtual bool NeedNewEdges () { return false; }
};
```

The prototype for the virtual constructor is given as:

```
typedef PMI_NonLocal_Recombination_Base*
    new_PMI_NonLocal_Recombination_Base_func
    (const PMI_Device_Environment& env);
```

## Chapter 39: Physical Model Interface

### Generation–Recombination

```
extern "C" new_PMI_NonLocal_Recombination_Base_func  
new_PMI_NonLocal_Recombination_Base;
```

## Example: Point-to-Point Tunneling

This example is presented in [Example: Point-to-Point Tunneling on page 1221](#).

---

## Tunneling Parameters

For the dynamic nonlocal path band-to-band tunneling model, the conduction and valence band offset energies and the electron and hole tunneling masses can be computed using a PMI (see [Dynamic Nonlocal Path Band-to-Band Tunneling Model on page 529](#)).

The syntax in the command file is:

```
Band2Band (  
    Model=NonlocalPath( PMIModel=pmi_model_name )  
)
```

You select the default tunneling parameters by omitting the `PMIModel` keyword:

```
Band2Band (  
    Model=NonlocalPath  
)
```

## Dependencies

The tunneling parameters can depend on:

`t` Lattice temperature [K]

The PMI model must compute:

`DcPath` Conduction band offset energy [eV]

`DvPath` Valence band offset energy [eV]

`m_c` Electron tunneling mass [ $m_0$ ]

`m_v` Hole tunneling mass [ $m_0$ ]

## Simplified C++ Interface

The following base class is declared in the file `PMI.h`:

```
class PMI_DNPBTBT_Base : public PMI_Vertex_Base {
public:
    class Input : public PMI_Vertex_Input_Base {
public:
    Input (const PMI_DNPBTBT_Base* base, const int vertex);

    pmi_float t; // lattice temperature
};

class Output {
public:
    pmi_float Dcpath; // conduction band offset energy
    pmi_float Dvpath; // valence band offset energy
    pmi_float m_c; // electron tunneling mass
    pmi_float m_v; // hole tunneling mass
};

PMI_DNPBTBT_Base (const PMI_Environment& env);
virtual ~PMI_DNPBTBT_Base ();

virtual void compute (const Input& input, Output& output) = 0;
};
```

The prototype for the virtual constructor is given as:

```
typedef PMI_DNPBTBT_Base* new_PMI_DNPBTBT_Base_func
extern "C" new_PMI_DNPBTBT_Base_func new_PMI_DNPBTBT_Base;
```

## Example: Constant Tunneling Parameters

A simple constant tunneling parameters model can be implemented as follows:

```
#include "PMI.h"

class BTBTPParameter : public PMI_DNPBTBT_Base {
friend class PMI_Device_Base;
public:
    BTBTPParameter (const PMI_Environment& env);
    ~BTBTPParameter ();

    void compute (const Input& input, Output& output);
};

BTBTPParameter::
BTBTPParameter (const PMI_Environment& env) :
    PMI_DNPBTBT_Base (env)
{}
```

## Chapter 39: Physical Model Interface

### Mobility

```
BTBTParameter::  
~BTBTParameter ()  
{  
}  
  
void BTBTParameter::  
compute (const Input& input, Output& output)  
{  
    output.m_c = 0.3;  
    output.m_v = 0.4;  
    output.Dcpath = 0.02;  
    output.Dvpath = 0.01;  
}  
  
extern "C"  
PMI_DNPBTBT_Base* new_PMI_DNPBTBT_Base (const PMI_Environment& env)  
{return new BTBTParameter (env);  
}
```

---

## Mobility

Sentaurus Device supports different types of PMI mobility model:

- Doping-dependent mobility
- Mobility degradation at interfaces
- High-field saturation

PMI and built-in models can be used simultaneously. See [Chapter 15 on page 385](#) for details about how the contributions of different models are combined. PMI mobility models support anisotropic calculations and can be evaluated along different crystallographic axes. The following enumeration type determines the axis:

```
enum PMI_AnisotropyType {  
    PMI_Isotropic,  
    PMI_Anisotropic  
};
```

The default is isotropic mobility. If anisotropic mobilities are activated in the command file, then the PMI mobility classes are also instantiated in the anisotropic direction.

This section presents the following PMIs:

- [Doping-Dependent Mobility](#)
- [Fitting Parameter for Ballistic Mobility](#)
- [High-Field Saturation](#)
- [High-Field Saturation With Two Driving Forces](#)

## Chapter 39: Physical Model Interface

### Mobility

- Mobility Degradation at Interfaces

---

## Doping-Dependent Mobility

A doping-dependent PMI model must account for both the constant mobility and doping-dependent mobility models discussed in [Mobility due to Phonon Scattering on page 386](#) and [Doping-Dependent Mobility Degradation on page 387](#).

## Dependencies

The constant mobility and doping-dependent mobility  $\mu_{\text{dop}}$  can depend on the following variables:

$t$  Lattice temperature [K]

$n$  Electron density [ $\text{cm}^{-3}$ ]

$p$  Hole density [ $\text{cm}^{-3}$ ]

The PMI model must compute the following results:

$m$  Mobility  $\mu_{\text{dop}}$  [ $\text{cm}^2 \text{V}^{-1} \text{s}^{-1}$ ]

In the case of the standard interface, the following derivatives must be computed as well:

$dmdn$  Derivative of  $\mu_{\text{dop}}$  with respect to  $n$  [ $\text{cm}^5 \text{V}^{-1} \text{s}^{-1}$ ]

$dmdp$  Derivative of  $\mu_{\text{dop}}$  with respect to  $p$  [ $\text{cm}^5 \text{V}^{-1} \text{s}^{-1}$ ]

$dmdt$  Derivative of  $\mu_{\text{dop}}$  with respect to  $t$  [ $\text{cm}^2 \text{V}^{-1} \text{s}^{-1} \text{K}^{-1}$ ]

In most cases, it is not necessary to compute the derivatives with respect to the dopant concentrations.

## Chapter 39: Physical Model Interface

### Mobility

However, to model random dopant fluctuations (see [Random Dopant Fluctuations on page 790](#)), the PMI model must override the functions that compute the following values:

dmdNa      Derivative of  $\mu_{\text{dop}}$  with respect to the acceptor concentration [ $\text{cm}^5 \text{V}^{-1} \text{s}^{-1}$ ]

dmdNd      Derivative of  $\mu_{\text{dop}}$  with respect to the donor concentration [ $\text{cm}^5 \text{V}^{-1} \text{s}^{-1}$ ]

## Standard C++ Interface

The following base class is declared in the file `PMIModels.h`:

```
class PMI_DopingDepMobility : public PMI_Vertex_Interface {  
  
private:  
    const PMI_AnisotropyType anisoType;  
  
public:  
    PMI_DopingDepMobility (const PMI_Environment& env,  
                           const PMI_AnisotropyType anisotype);  
    virtual ~PMI_DopingDepMobility ();  
  
    PMI_AnisotropyType AnisotropyType () const { return anisoType; }  
  
    virtual void Compute_m  
    (const double n, const double p,  
     const double t, double& m) = 0;  
  
    virtual void Compute_dmdn  
    (const double n, const double p,  
     const double t, double& dmdn) = 0;  
  
    virtual void Compute_dmdp  
    (const double n, const double p,  
     const double t, double& dmdp) = 0;  
  
    virtual void Compute_dmdt  
    (const double n, const double p,  
     const double t, double& dmdt) = 0;  
  
    virtual void Compute_dmdNa  
    (const double n, const double p,  
     const double t, double& dmdNa);  
  
    virtual void Compute_dmdNd  
    (const double n, const double p,  
     const double t, double& dmdNd);  
};
```

## Chapter 39: Physical Model Interface

### Mobility

Two virtual constructors are required for electron and hole mobilities:

```
typedef PMI_DopingDepMobility* new_PMI_DopingDepMobility_func
    (const PMI_Environment& env, const PMI_AnisotropyType anisotype);
extern "C" new_PMI_DopingDepMobility_func new_PMI_DopingDep_e_Mobility;
extern "C" new_PMI_DopingDepMobility_func new_PMI_DopingDep_h_Mobility;
```

## Simplified C++ Interface

The following base class is declared in the file `PMI.h`:

```
class PMI_DopingDepMobility_Base : public PMI_Vertex_Base {

public:
    class Input : public PMI_Vertex_Input_Base {
public:
    pmi_float n;           // electron density
    pmi_float p;           // hole density
    pmi_float t;           // lattice temperature
    pmi_float acceptor;   // total acceptor concentration
    pmi_float donor;       // total donor concentration
};

    class Output {
public:
    pmi_float m;           // doping-dependent mobility
};

    PMI_DopingDepMobility_Base (const PMI_Environment& env,
                                const PMI_AnisotropyType anisotype);
    virtual ~PMI_DopingDepMobility_Base ();

    PMI_AnisotropyType AnisotropyType () const;

    virtual void compute (const Input& input, Output& output) = 0;
};
```

The prototype for the virtual constructor is given as:

```
typedef PMI_DopingDepMobility_Base* new_PMI_DopingDepMobility_Base_func
    (const PMI_Environment& env, const PMI_AnisotropyType anisotype);
extern "C" new_PMI_DopingDepMobility_Base_func
    new_PMI_DopingDep_e_Mobility_Base;
extern "C" new_PMI_DopingDepMobility_Base_func
    new_PMI_DopingDep_h_Mobility_Base;
```

## Example: Masetti Doping-Dependent Mobility

The built-in Masetti model (see [Masetti Model on page 388](#)) can also be implemented as a PMI model:

```
#include "PMIModels.h"

class Masetti_DopingDepMobility : public PMI_DopingDepMobility {
protected:
    const double T0;
    double mumax, Exponent, mumini1, mumini2, mul, Pc, Cr, Cs, alpha,
           beta;

public:
    Masetti_DopingDepMobility (const PMI_Environment& env,
                               const PMI_AnisotropyType anisotype);
    ~Masetti_DopingDepMobility () {}

    void Compute_m
        (const double n, const double p,
         const double t, double& m);

    void Compute_dmdn
        (const double n, const double p,
         const double t, double& dmdn);

    void Compute_dmdp
        (const double n, const double p,
         const double t, double& dmdp);
    void Compute_dmdt
        (const double n, const double p,
         const double t, double& dmdt);
};

Masetti_DopingDepMobility::
Masetti_DopingDepMobility (const PMI_Environment& env,
                           const PMI_AnisotropyType anisotype) :
    PMI_DopingDepMobility (env, anisotype),
    T0 (300.0)
{
}

void Masetti_DopingDepMobility::
    Compute_m (const double n, const double p,
               const double t, double& m)
{
    const double mu_const = mumax * pow (t/T0, -Exponent);
    const double Ni = Max (ReadDoping (PMI_Donor) +
                           ReadDoping (PMI_Acceptor), 1.0);
    m = mumini1 * exp (-Pc / Ni) +
        (mu_const - mumini2) / (1.0 + pow (Ni / Cr, alpha)) -
        mul / (1.0 + pow (Cs / Ni, beta));
}
```

## Chapter 39: Physical Model Interface

### Mobility

```
void Masetti_DopingDepMobility::  
Compute_dmdn (const double n, const double p,  
              const double t, double& dmdn)  
{ dmdn = 0.0;  
}  
  
void Masetti_DopingDepMobility::  
Compute_dmdp (const double n, const double p,  
              const double t, double& dmdp)  
{ dmdp = 0.0;  
}  
  
void Masetti_DopingDepMobility::  
Compute_dmdt (const double n, const double p,  
              const double t, double& dmdt)  
{ const double Ni = Max (ReadDoping (PMI_Donor) +  
                         ReadDoping (PMI_Acceptor), 1.0);  
  dmdt = mumax * (-Exponent/T0) * pow (t/T0, -Exponent - 1.0) /  
         (1.0 + pow (Ni / Cr, alpha));  
}  
  
class Masetti_e_DopingDepMobility : public Masetti_DopingDepMobility {  
public:  
  Masetti_e_DopingDepMobility (const PMI_Environment& env,  
                               const PMI_AnisotropyType anisotype);  
  ~Masetti_e_DopingDepMobility () {}  
};  
  
Masetti_e_DopingDepMobility::  
Masetti_e_DopingDepMobility (const PMI_Environment& env,  
                           const PMI_AnisotropyType anisotype) :  
  Masetti_DopingDepMobility (env, anisotype)  
{ // default values  
  mumax = InitParameter ("mumax_e", 1417.0);  
  Exponent = InitParameter ("Exponent_e", 2.5);  
  mumin1 = InitParameter ("mumin1_e", 52.2);  
  mumin2 = InitParameter ("mumin2_e", 52.2);  
  mul = InitParameter ("mul_e", 43.4);  
  Pc = InitParameter ("Pc_e", 0.0);  
  Cr = InitParameter ("Cr_e", 9.68e16);  
  Cs = InitParameter ("Cs_e", 3.43e20);  
  alpha = InitParameter ("alpha_e", 0.680);  
  beta = InitParameter ("beta_e", 2.0);  
}  
  
class Masetti_h_DopingDepMobility : public Masetti_DopingDepMobility {  
public:  
  Masetti_h_DopingDepMobility (const PMI_Environment& env,  
                               const PMI_AnisotropyType anisotype);  
  ~Masetti_h_DopingDepMobility () {}
```

```

};

Masetti_h_DopingDepMobility::
Masetti_h_DopingDepMobility (const PMI_Environment& env,
                           const PMI_AnisotropyType anisotype) :
    Masetti_DopingDepMobility (env, anisotype)

{ // default values
    mumax = InitParameter ("mumax_h", 470.5);
    Exponent = InitParameter ("Exponent_h", 2.2);
    mumin1 = InitParameter ("mumin1_h", 44.9);
    mumin2 = InitParameter ("mumin2_h", 0.0);
    mul = InitParameter ("mul_h", 29.0);
    Pc = InitParameter ("Pc_h", 9.23e16);
    Cr = InitParameter ("Cr_h", 2.23e17);
    Cs = InitParameter ("Cs_h", 6.10e20);
    alpha = InitParameter ("alpha_h", 0.719);
    beta = InitParameter ("beta_h", 2.0);
}

extern "C"
PMI_DopingDepMobility* new_PMI_DopingDep_e_Mobility
    (const PMI_Environment& env, const PMI_AnisotropyType anisotype)
{ return new Masetti_e_DopingDepMobility (env, anisotype);
}

extern "C"
PMI_DopingDepMobility* new_PMI_DopingDep_h_Mobility
    (const PMI_Environment& env, const PMI_AnisotropyType anisotype)
{ return new Masetti_h_DopingDepMobility (env, anisotype);
}

```

# Fitting Parameter for Ballistic Mobility

The fitting parameter  $k_{300}$  that is accessible in the `BalMob` parameter set can be stress dependent and can be recalculated by any variables within the PMI model of [Mobility Stress Factor on page 1377](#).

The name of a parameter  $k_{300}$  PMI model is specified as a `BalMob` option in the command file. It should be specified as a `Factor` option as well:

```
Physics {
    Mobility (pmi_parameterk_model_name)
    Piezo (
        Model (
            Mobility (
                Factor (
                    pmi_parameterk_model_name
                    pmi_stress_model_name
                    [ChannelDirection=<n>]
                    [AutoOrientation | ParameterSetName="<pname>" ]
```

## Chapter 39: Physical Model Interface

### Mobility

```
        )
    )
}
}
```

## Dependencies

A ballistic mobility PMI model can depend on any global variables declared in the mobility stress factor PMI model with the keyword `extern`.

## Standard C++ Interface

The following base class is declared in the file `PMIModels.h`:

```
class PMI_EXTERNAL PMI_ParameterK : public PMI_Vertex_Interface {
public:
    PMI_ParameterK (const PMI_Environment& env);

    virtual ~PMI_ParameterK ();

    // methods to be implemented by user
    virtual void compute
        (double& parameter_k, int q) = 0;
};
```

One virtual constructor is required for these ballistic mobility factors:

```
typedef PMI_ParameterK* new_PMI_ParameterK_func
    (const PMI_Environment& env);
extern "C" PMI_ParameterK_func new_PMI_ParameterK;
```

## Simplified C++ Interface

The following base class is declared in the file `PMI.h`:

```
class PMI_ParameterK_Base : public PMI_Vertex_Base {
public:
    class Input : public PMI_Vertex_Input_Base {
public:
    Input (const PMI_ParameterK_Base* ParameterK_base, const int32_t
           vertex);
    //pmi_float t1; // lattice temperature for material1
    };
    class Output {
public:
    pmi_float parameter_k; // Schottky resistance
    };
    PMI_ParameterK_Base (const PMI_Environment& env);
    virtual ~PMI_ParameterK_Base ();
    virtual void compute (const Input& input1, Output& output) = 0;
};
```

## Chapter 39: Physical Model Interface

### Mobility

The prototype for the virtual constructor is given as:

```
typedef PMI_ParameterK_Base* new_PMI_ParameterK_Base_func
    (const PMI_Environment& env);
extern "C" PMI_ParameterK_Base_func
    new_PMI_ParameterK_Base;
```

## Example: Ballistic Mobility

This example illustrates the implementation of the `pmi_KVM_paraK` (see [Effective Stress Model on page 990](#)) using the standard C++ interface.

A variable, for example, `stress_k` should be declared in the `pmi_EffectiveStressModel` as a global variable:

```
double stress_k;
```

Then, the example of the `pmi_KVM_paraK` can be created:

```
#include "PMI.h"
#include "PMIModels.h"

extern double stress_k;
class PMIK: public PMI_ParameterK {
private:

public:
    PMIK(const PMI_Environment& env);
    ~PMIK();
    void compute(double& parameter_k, int q);
};

// constructor
PMIK::PMIK(const PMI_Environment& env):PMI_ParameterK(env) {

}

// destructor
PMIK::~PMIK() {
    // nothing to do here (nothing allocated)
}

void PMIK::compute(double& parameter_k, int q) {
    double stress[6];
    stress[0] = 1.e-6*ReadStress(PMI_StressXX);
    stress[1] = 1.e-6*ReadStress(PMI_StressYY);
    stress[2] = 1.e-6*ReadStress(PMI_StressZZ);
    stress[3] = 1.e-6*ReadStress(PMI_StressYZ);
    stress[4] = 1.e-6*ReadStress(PMI_StressXZ);
    stress[5] = 1.e-6*ReadStress(PMI_StressXY);
    double xModelFraction = ReadxMoleFraction();
    double DFSII = ReadDistanceFromSemiconductorInsulatorInterface();
    double x, y, z;
    ReadCoordinate (x, y, z);
```

## Chapter 39: Physical Model Interface

### Mobility

```
    parameter_k = stress_k*xModelFraction;
    return;
}

extern "C" PMI_ParameterK* new_PMI_ParameterK( const PMI_Environment&
env) {
    return new PMIK(env);
}
```

---

## High-Field Saturation

The high-field saturation model computes the final mobility  $\mu$  as a function of the low-field mobility  $\mu_{\text{low}}$  and the driving force  $F_{\text{hfs}}$  (see [High-Field Saturation Models on page 438](#)).

## Dependencies

The mobility  $\mu$  computed by a high-field mobility model can depend on the following variables:

pot	Electrostatic potential [V]
t	Lattice temperature [K]
n	Electron density [ $\text{cm}^{-3}$ ]
p	Hole density [ $\text{cm}^{-3}$ ]
ct	Carrier temperature [K]
mulow	Low-field mobility $\mu_{\text{low}}$ [ $\text{cm}^2 \text{V}^{-1} \text{s}^{-1}$ ]
F	Driving force [ $\text{V cm}^{-1}$ ]

### Note:

The carrier temperature  $ct$  represents the electron temperature during the evaluation of the model for electrons, and the hole temperature during the evaluation of the model for holes. The parameter  $ct$  is only defined for hydrodynamic simulations. Otherwise, the value of  $ct = 0$  is used.

## Chapter 39: Physical Model Interface

### Mobility

The PMI model must compute the following results:

$\mu$  Mobility  $\mu$  [ $\text{cm}^2 \text{V}^{-1} \text{s}^{-1}$ ]

In the case of the standard interface, the following derivatives must be computed as well:

$\text{dmudpot}$	Derivative of $\mu$ with respect to $\text{pot}$ [ $\text{cm}^2 \text{V}^{-2} \text{s}^{-1}$ ]
$\text{dmudn}$	Derivative of $\mu$ with respect to $n$ [ $\text{cm}^5 \text{V}^{-1} \text{s}^{-1}$ ]
$\text{dmudp}$	Derivative of $\mu$ with respect to $p$ [ $\text{cm}^5 \text{V}^{-1} \text{s}^{-1}$ ]
$\text{dmudt}$	Derivative of $\mu$ with respect to $t$ [ $\text{cm}^2 \text{V}^{-1} \text{s}^{-1} \text{K}^{-1}$ ]
$\text{dmudct}$	Derivative of $\mu$ with respect to $c_t$ [ $\text{cm}^2 \text{V}^{-1} \text{s}^{-1} \text{K}^{-1}$ ]
$\text{dmudmulow}$	Derivative of $\mu$ with respect to $\text{mulow}$ (1)
$\text{dmudF}$	Derivative of $\mu$ with respect to $F$ [ $\text{cm}^3 \text{V}^{-2} \text{s}^{-1}$ ]

In most cases, it is not necessary to compute the derivatives with respect to the dopant concentrations. However, to model random dopant fluctuations (see [Random Dopant Fluctuations on page 790](#)), the PMI model must override the functions that compute the following values:

$\text{dmudNa}$	Derivative of $\mu$ with respect to the acceptor concentration [ $\text{cm}^5 \text{V}^{-1} \text{s}^{-1}$ ]
$\text{dmudNd}$	Derivative of $\mu$ with respect to the donor concentration [ $\text{cm}^5 \text{V}^{-1} \text{s}^{-1}$ ]

## Standard C++ Interface

The enumeration type `PMI_HighFieldDrivingForce` describes the driving force as specified in the command file:

```
enum PMI_HighFieldDrivingForce {
    PMI_HighFieldParallelElectricField,
    PMI_HighFieldParallelToInterfaceElectricField,
    PMI_HighFieldGradQuasiFermi
};
```

## Chapter 39: Physical Model Interface

### Mobility

The following base class is declared in the file `PMIModels.h`:

```
class PMI_HighFieldMobility : public PMI_Vertex_Interface {  
  
private:  
    const PMI_HighFieldDrivingForce drivingForce;  
    const PMI_AnisotropyType anisoType;  
  
public:  
    PMI_HighFieldMobility (const PMI_Environment& env,  
                           const PMI_HighFieldDrivingForce force,  
                           const PMI_AnisotropyType anisotype);  
  
    virtual ~PMI_HighFieldMobility ();  
  
    PMI_HighFieldDrivingForce HighFieldDrivingForce () const;  
    PMI_AnisotropyType AnisotropyType () const;  
  
    virtual void Compute_mu  
        (const double pot, const double n,  
         const double p, const double t, const double ct,  
         const double mulow, const double F, double& mu) = 0;  
  
    virtual void Compute_dmudpot  
        (const double pot, const double n,  
         const double p, const double t, const double ct,  
         const double mulow, const double F, double& dmudpot) = 0;  
  
    virtual void Compute_dmudn  
        (const double pot, const double n,  
         const double p, const double t, const double ct,  
         const double mulow, const double F, double& dmudn) = 0;  
  
    virtual void Compute_dmudp  
        (const double pot, const double n,  
         const double p, const double t, const double ct,  
         const double mulow, const double F, double& dmudp) = 0;  
  
    virtual void Compute_dmudt  
        (const double pot, const double n,  
         const double p, const double t, const double ct,  
         const double mulow, const double F, double& dmudt) = 0;  
  
    virtual void Compute_dmudct  
        (const double pot, const double n,  
         const double p, const double t, const double ct,  
         const double mulow, const double F, double& dmudct) = 0;  
  
    virtual void Compute_dmudmulow  
        (const double pot, const double n,  
         const double p, const double t, const double ct,  
         const double mulow, const double F, double& dmudmulow) = 0;
```

## Chapter 39: Physical Model Interface

### Mobility

```
virtual void Compute_dmudF
  (const double pot, const double n,
   const double p, const double t, const double ct,
   const double mulow, const double F, double& dmudF) = 0;

virtual void Compute_dmudNa
  (const double pot, const double n,
   const double p, const double t, const double ct,
   const double mulow, const double F, double& dmudNa);

virtual void Compute_dmudNd
  (const double pot, const double n,
   const double p, const double t, const double ct,
   const double mulow, const double F, double& dmudNd);
};


```

Two virtual constructors are required for electron and hole mobilities:

```
typedef PMI_HighFieldMobility* new_PMI_HighFieldMobility_func
  (const PMI_Environment& env, const PMI_HighFieldDrivingForce force,
   const PMI_AnisotropyType anisotype);
extern "C" new_PMI_HighFieldMobility_func
  new_PMI_HighField_e_Mobility;
extern "C" new_PMI_HighFieldMobility_func
  new_PMI_HighField_h_Mobility;
```

## Simplified C++ Interface

The following base class is declared in the file `PMI.h`:

```
class PMI_HighFieldMobility_Base : public PMI_Vertex_Base {

public:
  class Input : public PMI_Vertex_Input_Base {
public:
  pmi_float pot;           // electrostatic potential
  pmi_float n;             // electron density
  pmi_float p;             // hole density
  pmi_float t;             // lattice temperature
  pmi_float ct;            // carrier temperature
  pmi_float mulow;         // low field mobility
  pmi_float F;              // driving force
  pmi_float acceptor;      // total acceptor concentration
  pmi_float donor;          // total donor concentration
};

  class Output {
public:
  pmi_float mu;             // mobility
};

  PMI_HighFieldMobility_Base (const PMI_Environment& env,
                           const PMI_HighFieldDrivingForce force,
```

## Chapter 39: Physical Model Interface

### Mobility

```
        const PMI_AnisotropyType anisotype);
virtual ~PMI_HighFieldMobility_Base () ;

PMI_HighFieldDrivingForce HighFieldDrivingForce () const;
PMI_AnisotropyType AnisotropyType () const;

virtual void compute (const Input& input, Output& output) = 0;
};
```

The prototype for the virtual constructor is given as:

```
typedef PMI_HighFieldMobility_Base*
    new_PMI_HighFieldMobility_Base_func
(const PMI_Environment& env, const PMI_HighFieldDrivingForce force,
 const PMI_AnisotropyType anisotype);
extern "C" new_PMI_HighFieldMobility_Base_func
new_PMI_HighField_e_Mobility_Base;
extern "C" new_PMI_HighFieldMobility_Base_func
new_PMI_HighField_h_Mobility_Base;
```

## Example: Canali High-Field Saturation

This example presents the PMI implementation of the Canali model:

$$\mu = \frac{\mu_{\text{low}}}{\left[ 1 + \frac{\mu_{\text{low}} F_{\text{hfs}}}{v_{\text{sat}}} \beta \right]^{1/\beta}} \quad (1337)$$

where:

- $\beta = \beta_0 \frac{T}{T_0}^{\beta_{\text{exp.}}}$  and  $v_{\text{sat}} = v_{\text{sat0}} \frac{T}{T_0}^{v_{\text{sat,exp}}}$

The built-in Canali model is discussed in [Extended Canali Model on page 440](#).

```
#include "PMIModels.h"

class Canali_HighFieldMobility : public PMI_HighFieldMobility {

private:
    double beta, vsat, Fabs, val, valb, valb1, valb1b;
    void Compute_internal (const double t, const double mulow,
                           const double F);

protected:
    const double T0;
    double beta0, betaexp, vsat0, vsatexp;

public:
    Canali_HighFieldMobility (const PMI_Environment& env,
                             const PMI_HighFieldDrivingForce force,
                             const PMI_AnisotropyType anisotype);
```

## Chapter 39: Physical Model Interface

### Mobility

```
~Canali_HighFieldMobility ();

void Compute_mu
  (const double pot, const double n,
   const double p, const double t, const double ct,
   const double mulow, const double F,
   double& mu);

void Compute_dmudpot
  (const double pot, const double n,
   const double p, const double t, const double ct,
   const double mulow, const double F, double& dmudpot);

void Compute_dmudn
  (const double pot, const double n,
   const double p, const double t, const double ct,
   const double mulow, const double F, double& dmudn);

void Compute_dmudp
  (const double pot, const double n,
   const double p, const double t, const double ct,
   const double mulow, const double F, double& dmudp);

void Compute_dmudt
  (const double pot, const double n,
   const double p, const double t, const double ct,
   const double mulow, const double F, double& dmudt);

void Compute_dmudct
  (const double pot, const double n,
   const double p, const double t, const double ct,
   const double mulow, const double F, double& dmudct);

void Compute_dmudmulow
  (const double pot, const double n,
   const double p, const double t, const double ct,
   const double mulow, const double F, double& dmudmulow);

void Compute_dmudF
  (const double pot, const double n,
   const double p, const double t, const double ct,
   const double mulow, const double F, double& dmudF);
};

void Canali_HighFieldMobility::
Compute_internal (const double t, const double mulow, const double F)
{ beta = beta0 * pow (t/T0, betaexp);
  vsat = vsat0 * pow (t/T0, -vsatexp);
  Fabs = fabs (F);
  val = mulow * Fabs / vsat;
  valb = pow (val, beta);
```

## Chapter 39: Physical Model Interface

### Mobility

```
valb1 = 1.0 + valb;
valb1b = pow (valb1, 1.0/beta);
}

Canali_HighFieldMobility::  
Canali_HighFieldMobility (const PMI_Environment& env,
                           const PMI_HighFieldDrivingForce force,
                           const PMI_AnisotropyType anisotype) :
    PMI_HighFieldMobility (env, force, anisotype),
    T0 (300.0)
{
}

Canali_HighFieldMobility::  
~Canali_HighFieldMobility ()
{
}

void Canali_HighFieldMobility::  
Compute_mu (const double pot, const double n,
            const double p, const double t, const double ct,
            const double mulow, const double F, double& mu)
{ Compute_internal (t, mulow, F);
  mu = mulow / valb1b;
}

void Canali_HighFieldMobility::  
Compute_dmudpot (const double pot, const double n,
                  const double p, const double t, const double ct,
                  const double mulow, const double F, double& dmudpot)
{ dmudpot = 0.0;
}

void Canali_HighFieldMobility::  
Compute_dmudn (const double pot, const double n,
                const double p, const double t, const double ct,
                const double mulow, const double F, double& dmudn)
{ dmudn = 0.0;
}

void Canali_HighFieldMobility::  
Compute_dmudp (const double pot, const double n,
                const double p, const double t, const double ct,
                const double mulow, const double F, double& dmudp)
{ dmudp = 0.0;
}

void Canali_HighFieldMobility::  
Compute_dmudt (const double pot, const double n,
                const double p, const double t, const double ct,
                const double mulow, const double F, double& dmudt)
{ Compute_internal (t, mulow, F);
```

## Chapter 39: Physical Model Interface

### Mobility

```
const double mu = mulow / valb11b;
const double dmudbeta = mu * (log (valb1) / (beta*beta) -
                               valb * log (val) / (beta * valb1));
const double dmudvsat = (mu * valb) / (valb1 * vsat);
const double dbetadt = beta * betaxp / t;
const double dvsatdt = -vsat * vsatxp / t;
dmudt = dmudbeta * dbetadt + dmudvsat * dvsatdt;
}

void Canali_HighFieldMobility:::
Compute_dmudct (const double pot, const double n,
                  const double p, const double t, const double ct,
                  const double mulow, const double F, double& dmudct)
{ dmudct = 0.0;
}

void Canali_HighFieldMobility:::
Compute_dmudmulow (const double pot, const double n,
                     const double p, const double t, const double ct,
                     const double mulow, const double F, double&
                     dmudmulow)
{ Compute_internal (t, mulow, F);
  dmudmulow = 1.0 / (valb1 * valb11b);
}

void Canali_HighFieldMobility:::
Compute_dmudF (const double pot, const double n,
                const double p, const double t, const double ct,
                const double mulow, const double F, double& dmudF)
{ Compute_internal (t, mulow, F);
  const double mu = mulow / valb11b;
  const double signF = (F >= 0.0) ? 1.0 : -1.0;
  dmudF = -mu * pow (mulow/vsat, beta) * pow (fabs, beta-1.0) *
          signF / valb1;
}

class Canali_e_HighFieldMobility : public Canali_HighFieldMobility {
public:
  Canali_e_HighFieldMobility (const PMI_Environment& env,
                             const PMI_HighFieldDrivingForce force,
                             const PMI_AnisotropyType anisotype);

  ~Canali_e_HighFieldMobility () {}

  Canali_e_HighFieldMobility:::
  Canali_e_HighFieldMobility (const PMI_Environment& env,
                             const PMI_HighFieldDrivingForce force,
                             const PMI_AnisotropyType anisotype) :
    Canali_HighFieldMobility (env, force, anisotype)
  { // default values
    beta0 = InitParameter ("beta0_e", 1.109);
    betaxp = InitParameter ("betaexp_e", 0.66);
```

## Chapter 39: Physical Model Interface

### Mobility

```
vsat0 = InitParameter ("vsat0_e", 1.07e7);
vsatexp = InitParameter ("vsatexp_e", 0.87);
}

class Canali_h_HighFieldMobility : public Canali_HighFieldMobility {
public:
    Canali_h_HighFieldMobility (const PMI_Environment& env,
                                const PMI_HighFieldDrivingForce force,
                                const PMI_AnisotropyType anisotype);

    ~Canali_h_HighFieldMobility () {}

Canali_h_HighFieldMobility:::
Canali_h_HighFieldMobility (const PMI_Environment& env,
                           const PMI_HighFieldDrivingForce force,
                           const PMI_AnisotropyType anisotype) :
    Canali_HighFieldMobility (env, force, anisotype)
{ // default values
    beta0 = InitParameter ("beta0_h", 1.213);
    betaexp = InitParameter ("betaexp_h", 0.17);
    vsat0 = InitParameter ("vsat0_h", 8.37e6);
    vsatexp = InitParameter ("vsatexp_h", 0.52);
}

extern "C"
PMI_HighFieldMobility* new_PMI_HighField_e_Mobility
    (const PMI_Environment& env, const PMI_HighFieldDrivingForce force,
     const PMI_AnisotropyType anisotype)
{ return new Canali_e_HighFieldMobility (env, force, anisotype);
}

extern "C"
PMI_HighFieldMobility* new_PMI_HighField_h_Mobility
    (const PMI_Environment& env, const PMI_HighFieldDrivingForce force,
     const PMI_AnisotropyType anisotype)
{ return new Canali_h_HighFieldMobility (env, force, anisotype);
}
```

---

## High-Field Saturation With Two Driving Forces

This PMI allows you to compute the final mobility  $\mu$  as a function of the low-field mobility  $\mu_{\text{low}}$  and two driving fields (see [High-Field Saturation Models on page 438](#)). One field is the gradient of the quasi-Fermi energy; the other is derived from the electric field (see [Driving Force Models on page 447](#)).

## Command File

The model is specified using `PMIModel` as an option to `HighFieldSaturation`, `eHighFieldSaturation`, or `hHighFieldSaturation`. The name of the model must be

## Chapter 39: Physical Model Interface

### Mobility

provided with the `Name` parameter of `PMIModel`. Optionally, an index and a string can be specified, which will be passed to and interpreted by the model:

```
eMobility(  
    HighFieldSaturation(  
        PMIModel (  
            Name = <string>  
            Index = <int>  
            String = <string>)  
        EparallelToInterface | Eparallel | ElectricField)  
)
```

## Dependencies

The high-field mobility can depend on the following variables:

<code>mulow</code>	Low-field mobility $\mu_{\text{low}}$ [ $\text{cm}^2 \text{V}^{-1} \text{s}^{-1}$ ]
<code>n</code>	Electron density [ $\text{cm}^{-3}$ ]
<code>p</code>	Hole density [ $\text{cm}^{-3}$ ]
<code>T</code>	Lattice temperature [K]
<code>cT</code>	Carrier temperature [K]
<code>Epar</code>	Modulus of electric field driving force [ $\text{V cm}^{-1}$ ]
<code>gradQF</code>	Modulus of gradient of the quasi-Fermi energy [ $\text{eV cm}^{-1}$ ]
<code>EprodQF</code>	Scalar product of electric field driving force and gradient of quasi-Fermi energy [ $\text{eV V cm}^{-2}$ ]
<code>Na0</code>	Acceptor concentration [ $\text{cm}^{-3}$ ]
<code>Nd0</code>	Donor concentration [ $\text{cm}^{-3}$ ]

The electric field used to obtain `Epar` is determined by `EparallelToInterface`, `Eparallel`, or `ElectricField` as for other high-field mobility models (see [Driving Force Models on page 447](#)). With `EparallelToInterface`, the electric field used to compute `EprodQF` is the projection of the electric field parallel to the interface; otherwise, the full electric field is used to obtain `EprodQF`.

## Chapter 39: Physical Model Interface

### Mobility

The PMI model must compute the following results:

`val`    High-field mobility  $\mu$  [ $\text{cm}^2 \text{V}^{-1} \text{s}^{-1}$ ]

In the case of the standard interface, the following derivatives must be computed as well:

<code>dval_dmulow</code>	Derivative of $\mu$ with respect to <code>mulow</code> [1]
<code>dval_dn</code>	Derivative of $\mu$ with respect to <code>n</code> [ $\text{cm}^5 \text{V}^{-1} \text{s}^{-1}$ ]
<code>dval_dp</code>	Derivative of $\mu$ with respect to <code>p</code> [ $\text{cm}^5 \text{V}^{-1} \text{s}^{-1}$ ]
<code>dval_dT</code>	Derivative of $\mu$ with respect to <code>T</code> [ $\text{cm}^2 \text{V}^{-1} \text{s}^{-1} \text{K}^{-1}$ ]
<code>dval_dcT</code>	Derivative of $\mu$ with respect to <code>cT</code> [ $\text{cm}^2 \text{V}^{-1} \text{s}^{-1} \text{K}^{-1}$ ]
<code>dval_dEpar</code>	Derivative of $\mu$ with respect to <code>Epar</code> [ $\text{cm}^3 \text{V}^{-2} \text{s}^{-1}$ ]
<code>dval_dgradQF</code>	Derivative of $\mu$ with respect to <code>gradQF</code> [ $\text{cm}^3 \text{eV}^{-1} \text{V}^{-1} \text{s}^{-1}$ ]
<code>dval_dEprodQF</code>	Derivative of $\mu$ with respect to <code>EprodQF</code> [ $\text{cm}^4 \text{eV}^{-1} \text{V}^{-2} \text{s}^{-1}$ ]
<code>dval_dNa0</code>	Derivative of $\mu$ with respect to <code>Na0</code> [ $\text{cm}^5 \text{V}^{-1} \text{s}^{-1}$ ]
<code>dval_dNd0</code>	Derivative of $\mu$ with respect to <code>Nd0</code> [ $\text{cm}^5 \text{V}^{-1} \text{s}^{-1}$ ]

## Standard C++ Interface

The PMI base class is declared in `PMIModels.h` as:

```
class PMI_HighFieldMobility2 : public PMI_Vertex_Interface {  
  
public:  
    // the input data coming from the simulator  
    class idata {  
public:  
        double mulow() const;           // low-field mobility  
        double n() const;              // electron density  
        double p() const;              // hole density  
        double T() const;              // lattice temperature  
        double cT() const;             // carrier temperature  
        double Epar() const;            // parallel electric field  
        double gradQF() const;          // gradient of quasi-Fermi energy
```

## Chapter 39: Physical Model Interface

### Mobility

```
    double EprodQF() const;      // product gradient QF and electric
                                // field
    double Na0() const;         // acceptor concentration
    double Nd0() const;         // donor concentration
};

// the results computed by the PMI
class odata {
public:
    double& val();           // mobility
    double& dval_dmulow();    // derivative wrt. low-field mobility
    double& dval_dn();        // wrt. electron density
    double& dval_dp();        // wrt. hole density
    double& dval_dT();        // wrt. lattice temperature
    double& dval_dcT();       // wrt. carrier temperature
    double& dval_dEpar();     // wrt. parallel electric field
    double& dval_dgradQF();   // wrt. gradient of quasi-Fermi energy
    double& dval_dEprodQF();  // wrt. product gradient QF and field
    double& dval_dNa0();      // wrt. acceptor concentration
    double& dval_dNd0();      // wrt. donor concentration
};

// constructor and destructor
PMI_HighFieldMobility2(const PMI_Environment& env,
                       const int model_index,
                       const std::string& model_string,
                       const PMI_AnisotropyType anisotype);
virtual ~PMI_HighFieldMobility2();

PMI_AnisotropyType AnisotropyType () const;

// compute value and derivatives
virtual void compute(const idata* id, odata* od) = 0;
};
```

The `compute` function receives its input from `id`. It returns the results using `od` by assignment using the member functions of `od`. The framework initializes the values of the derivatives to zero, so you only have to compute derivatives for variables the PMI actually uses.

The following virtual constructor must be implemented:

```
typedef PMI_HighFieldMobility2* new_PMI_HighFieldMobility2_func
(const PMI_Environment& env,
 int model_index,
 const std::string& model_string,
 const PMI_AnisotropyType anisotype);
extern "C" new_PMI_HighFieldMobility2_func new_PMI_HighFieldMobility2;
```

## Simplified C++ Interface

The following base class is declared in `PMI.h`:

```
class PMI_HighFieldMobility2_Base : public PMI_Vertex_Base {

public:
    class Input : public PMI_Vertex_Input_Base {
public:
    Input (const PMI_HighFieldMobility2_Base*
           highfieldmobility2_base, const int vertex);
    pmi_float mulow;          // low-field mobility
    pmi_float n;              // electron density
    pmi_float p;              // hole density
    pmi_float T;              // lattice temperature
    pmi_float cT;             // carrier temperature
    pmi_float Epar;           // parallel electric field
    pmi_float gradQF;         // gradient of quasi-Fermi energy
    pmi_float EprodQF;        // product gradient QF and electric field
    pmi_float Na0;            // acceptor concentration
    pmi_float Nd0;            // donor concentration
};

class Output {
public:
    pmi_float val;           // mobility
};

PMI_HighFieldMobility2_Base (const PMI_Environment& env,
                           const int model_index,
                           const std::string& model_string,
                           const PMI_AnisotropyType anisotype);
virtual ~PMI_HighFieldMobility2_Base ();

int model_index () const;
const std::string& model_string () const;
PMI_AnisotropyType AnisotropyType () const;

virtual void compute (const Input& input, Output& output) = 0;
};
```

The following virtual constructor is provided:

```
typedef PMI_HighFieldMobility2_Base*
        new_PMI_HighFieldMobility2_Base_func
(const PMI_Environment& env, const int model_index,
 const std::string& model_string,
 const PMI_AnisotropyType anisotype);
extern "C" new_PMI_HighFieldMobility2_Base_func
new_PMI_HighField_Mobility2_Base;
```

---

## Mobility Degradation at Interfaces

Sentaurus Device uses Matthiessen's rule:

$$\frac{1}{\mu} = \frac{1}{\mu_{\text{dop}}} + \frac{1}{\mu_{\text{enormal}}} \quad (1338)$$

to combine the constant and doping-dependent mobility  $\mu_{\text{dop}}$ , and the surface contribution  $\mu_{\text{enormal}}$  (see [Mobility Degradation at Interfaces on page 403](#)). To express no mobility degradation, for example, in the bulk of a device, it is necessary to set  $\mu_{\text{enormal}} = \infty$ . To avoid numeric difficulties, the PMI requires the calculation of the inverse mobility  $1/\mu_{\text{enormal}}$  instead of  $\mu_{\text{enormal}}$ .

As an additional precaution, Sentaurus Device does not evaluate the PMI model if the normal electric field  $F_{\perp}$  is less than `EnormMinimum`, where `EnormMinimum` is a parameter that can be specified in the `PMI_model_name` section of the parameter file. By default, `EnormMinimum = 1 V/cm` is used.

## Dependencies

The mobility degradation at interfaces can depend on the following variables:

`dist` Distance to nearest interface [cm]

`pot` Electrostatic potential [V]

`enorm` Normal electric field [ $\text{V cm}^{-1}$ ]

`t` Lattice temperature [K]

`n` Electron density [ $\text{cm}^{-3}$ ]

`p` Hole density [ $\text{cm}^{-3}$ ]

`ct` Carrier temperature [K]

### Note:

If Sentaurus Device cannot determine the distance to the nearest interface, then the value of `dist = 1010` is used.

The carrier temperature `ct` represents the electron temperature during the evaluation of the model for electrons, and the hole temperature during the evaluation of the model for holes. The parameter `ct` is only defined for hydrodynamic simulations. Otherwise, the value of `ct = 0` is used.

## Chapter 39: Physical Model Interface

### Mobility

The PMI model must compute the following results:

`muinv`      Inverse of mobility  $1/\mu_{\text{enormal}}$  [ $\text{cm}^{-2}\text{Vs}$ ]

In the case of the standard interface, the following derivatives must be computed as well:

`dmuinvdpot`      Derivative of  $1/\mu_{\text{enormal}}$  with respect to `pot` [ $\text{cm}^{-2}\text{s}$ ]

`dmuinvdenorm`      Derivative of  $1/\mu_{\text{enormal}}$  with respect to `enorm` [ $\text{cm}^{-1}\text{s}$ ]

`dmuinvdn`      Derivative of  $1/\mu_{\text{enormal}}$  with respect to `n` [ $\text{cm}^3\text{Vs}$ ]

`dmuinvdp`      Derivative of  $1/\mu_{\text{enormal}}$  with respect to `p` [ $\text{cm}^3\text{Vs}$ ]

`dmuinvdt`      Derivative of  $1/\mu_{\text{enormal}}$  with respect to `t` [ $\text{cm}^{-2}\text{VsK}^{-1}$ ]

`dmuinvdct`      Derivative of  $1/\mu_{\text{enormal}}$  with respect to `ct` [ $\text{cm}^{-2}\text{VsK}^{-1}$ ]

In most cases, it is not necessary to compute the derivatives with respect to the dopant concentrations. However, to model random dopant fluctuations (see [Random Dopant Fluctuations on page 790](#)), the PMI model must override the functions that compute the following values:

`dmuinvdNa`      Derivative of  $1/\mu_{\text{enormal}}$  with respect to the acceptor concentration [ $\text{cm}^3\text{Vs}$ ]

`dmuinvdNd`      Derivative of  $1/\mu_{\text{enormal}}$  with respect to the donor concentration [ $\text{cm}^3\text{Vs}$ ]

## Standard C++ Interface

The enumeration type `PMI_EnormalType` describes the type of the normal electric field  $F_{\perp}$ :

```
enum PMI_EnormalType {
    PMI_EnormalToCurrent,
    PMI_EnormalToInterface
};
```

## Chapter 39: Physical Model Interface

### Mobility

The following base class is declared in the file PMIModels.h:

```
class PMI_EnormalMobility : public PMI_Vertex_Interface {  
  
public:  
    PMI_EnormalMobility (const PMI_Environment& env,  
                         const PMI_EnormalType type,  
                         const PMI_AnisotropyType anisotype);  
    virtual ~PMI_EnormalMobility ();  
  
    PMI_EnormalType EnormalType () const;  
    PMI_AnisotropyType AnisotropyType () const;  
  
    virtual void Compute_muinv  
        (const double dist, const double pot,  
         const double enorm, const double n, const double p,  
         const double t, const double ct, double& muinv) = 0;  
  
    virtual void Compute_dmuinvdpot  
        (const double dist, const double pot,  
         const double enorm, const double n, const double p,  
         const double t, const double ct, double& dmuinvdpot) = 0;  
  
    virtual void Compute_dmuinvdenorm  
        (const double dist, const double pot,  
         const double enorm, const double n, const double p,  
         const double t, const double ct, double& dmuinvdenorm) = 0;  
  
    virtual void Compute_dmuinvdn  
        (const double dist, const double pot,  
         const double enorm, const double n, const double p,  
         const double t, const double ct, double& dmuinvdn) = 0;  
  
    virtual void Compute_dmuinvdp  
        (const double dist, const double pot,  
         const double enorm, const double n, const double p,  
         const double t, const double ct, double& dmuinvdp) = 0;  
  
    virtual void Compute_dmuinvdt  
        (const double dist, const double pot,  
         const double enorm, const double n, const double p,  
         const double t, const double ct, double& dmuinvdt) = 0;  
  
    virtual void Compute_dmuinvdct  
        (const double dist, const double pot,  
         const double enorm, const double n, const double p,  
         const double t, const double ct, double& dmuinvdct) = 0;  
  
    virtual void Compute_dmuinvdNa  
        (const double dist, const double pot,  
         const double enorm, const double n, const double p,  
         const double t, const double ct, double& dmuinvdNa);
```

## Chapter 39: Physical Model Interface

### Mobility

```
virtual void Compute_dmuinvdNd
  (const double dist, const double pot,
   const double enorm, const double n, const double p,
   const double t, const double ct, double& dmuinvdNd);
};
```

Two virtual constructors are required for electron and hole mobilities:

```
typedef PMI_EnormalMobility* new_PMI_EnormalMobility_func
  (const PMI_Environment& env, const PMI_EnormalType type,
   const PMI_AnisotropyType anisotype);
extern "C" new_PMI_EnormalMobility_func new_PMI_Enormal_e_Mobility;
extern "C" new_PMI_EnormalMobility_func new_PMI_Enormal_h_Mobility;
```

## Simplified C++ Interface

The following base class is declared in the file `PMI.h`:

```
class PMI_EnormalMobility_Base : public PMI_Vertex_Base {

public:
  class Input : public PMI_Vertex_Input_Base {
public:
  pmi_float dist;           // distance to nearest interface
  pmi_float pot;            // electrostatic potential
  pmi_float enorm;          // normal electric field
  pmi_float n;               // electron density
  pmi_float p;               // hole density
  pmi_float t;               // lattice temperature
  pmi_float ct;              // carrier temperature
  pmi_float acceptor;        // total acceptor concentration
  pmi_float donor;            // total donor concentration
};

  class Output {
public:
  pmi_float muinv;          // inverse of mobility degradation
};

  PMI_EnormalMobility_Base (const PMI_Environment& env,
                           const PMI_EnormalType type,
                           const PMI_AnisotropyType anisotype);
  virtual ~PMI_EnormalMobility_Base ();

  PMI_EnormalType EnormalType () const;
  PMI_AnisotropyType AnisotropyType () const;

  virtual void compute (const Input& input, Output& output) = 0;
};
```

## Chapter 39: Physical Model Interface

### Mobility

The prototype for the virtual constructor is given as:

```
typedef PMI_EnormalMobility_Base* new_PMI_EnormalMobility_Base_func
    (const PMI_Environment& env, const PMI_EnormalType type,
     const PMI_AnisotropyType anisotype);
extern "C" new_PMI_EnormalMobility_Base_func
    new_PMI_Enormal_e_Mobility_Base;
extern "C" new_PMI_EnormalMobility_Base_func
    new_PMI_Enormal_h_Mobility_Base;
```

## Example: Lombardi Mobility Degradation

This example illustrates the implementation of a slightly simplified Lombardi model (see [Mobility Degradation at Interfaces on page 403](#)) using the PMI. The contribution due to acoustic phonon-scattering has the form:

$$\mu_{ac} = \frac{B}{F_\perp} + \frac{CN_i^\lambda}{F_\perp^{1/3}(T/T_0)} \quad (1339)$$

where  $T_0 = 300$  K.

The contribution due to surface roughness scattering is given by:

$$\mu_{sr} = \frac{F_\perp^2}{\delta} + \frac{F_\perp^3}{\eta}^{-1} \quad (1340)$$

The mobilities  $\mu_{ac}$  and  $\mu_{sr}$  are combined according to Matthiessen's rule with an additional damping factor:

$$\frac{1}{\mu_{enormal}} = e^{-\frac{l}{l_{crit}}} \cdot \frac{1}{\mu_{ac}} + \frac{1}{\mu_{sr}} \quad (1341)$$

where  $l$  is the distance to the nearest semiconductor–insulator interface point:

```
#include "PMIModels.h"

class Lombardi_EnormalMobility : public PMI_EnormalMobility {

protected:
    const double T0;
    double B, C, lambda, delta, eta, l_crit;

public:
    Lombardi_EnormalMobility (const PMI_Environment& env,
                             const PMI_EnormalType type,
                             const PMI_AnisotropyType anisotype);

    ~Lombardi_EnormalMobility ();

    void Compute_muinv
        (const double dist, const double pot,
```

## Chapter 39: Physical Model Interface

### Mobility

```
const double enorm, const double n, const double p,
const double t, const double ct, double& muinv);

void Compute_dmuinvdpot
(const double dist, const double pot,
const double enorm, const double n, const double p,
const double t, const double ct, double& dmuinvdpot);

void Compute_dmuinvdenorm
(const double dist, const double pot,
const double enorm, const double n, const double p,
const double t, const double ct, double& dmuinvdenorm);

void Compute_dmuinvdn
(const double dist, const double pot,
const double enorm, const double n, const double p,
const double t, const double ct, double& dmuinvdn);

void Compute_dmuinvdp
(const double dist, const double pot,
const double enorm, const double n, const double p,
const double t, const double ct, double& dmuinvdp);

void Compute_dmuinvd
(const double dist, const double pot,
const double enorm, const double n, const double p,
const double t, const double ct, double& dmuinvd);

void Compute_dmuinvdct
(const double dist, const double pot,
const double enorm, const double n, const double p,
const double t, const double ct, double& dmuinvdct);
};

Lombardi_EnormalMobility:::
Lombardi_EnormalMobility (const PMI_Environment& env,
                           const PMI_EnormalType type,
                           const PMI_AnisotropyType anisotype) :
    PMI_EnormalMobility (env, type, anisotype),
    T0 (300.0)
{
}

Lombardi_EnormalMobility:::
~Lombardi_EnormalMobility ()
{
}

void Lombardi_EnormalMobility:::
Compute_muinv (const double dist, const double pot,
                const double enorm, const double n, const double p,
                const double t, const double ct, double& muinv)
```

## Chapter 39: Physical Model Interface

### Mobility

```
{ const double Ni = ReadDoping (PMI_Donor) + ReadDoping
    (PMI_Acceptor);
const double denom_ac_inv =
    B + pow (enorm, 2.0/3.0) * C * pow (Ni, lambda) * T0 / t;
const double mu_ac_inv = enorm / denom_ac_inv;
const double mu_sr_inv = enorm * enorm / delta +
    pow (enorm, 3.0) / eta;
const double damping = exp (-dist/l_crit);
muinv = damping * (mu_ac_inv + mu_sr_inv);
}

void Lombardi_EnormalMobility::
Compute_dmuinvdpot (const double dist, const double pot,
                     const double enorm, const double n,
                     const double p, const double t, const double ct,
                     double& dmuinvdpot)
{ dmuinvdpot = 0.0;
}

void Lombardi_EnormalMobility::
Compute_dmuinvdenorm (const double dist, const double pot,
                      const double enorm, const double n,
                      const double p, const double t,
                      const double ct, double& dmuinvdenorm)
{ const double Ni = ReadDoping (PMI_Donor) + ReadDoping
    (PMI_Acceptor);
const double denom_ac_inv =
    B + pow (enorm, 2.0/3.0) * C * pow (Ni, lambda) * T0 / t;
const double dmu_ac_inv_denorm =
    (2.0 * B + denom_ac_inv) / (3.0 * denom_ac_inv * denom_ac_inv);
const double mu_sr_inv_denorm =
    2.0 * enorm / delta + 3.0 * enorm * enorm / eta;
const double damping = exp (-dist/l_crit);
dmuinvdnorm = damping * (dmu_ac_inv_denorm + mu_sr_inv_denorm);
}

void Lombardi_EnormalMobility::
Compute_dmuinvdn (const double dist, const double pot,
                  const double enorm, const double n, const double p,
                  const double t, const double ct, double& dmuinvdn)
{ dmuinvdn = 0.0;
}

void Lombardi_EnormalMobility::
Compute_dmuinvdp (const double dist, const double pot,
                  const double enorm, const double n, const double p,
                  const double t, const double ct, double& dmuinvdp)
{ dmuinvdp = 0.0;
}

void Lombardi_EnormalMobility::
Compute_dmuinvdt (const double dist, const double pot,
                  const double enorm, const double n, const double p,
```

## Chapter 39: Physical Model Interface

### Mobility

```
        const double t, const double ct, double& dmuinvdt)
{ const double Ni = ReadDoping (PMI_Donor) + ReadDoping
    (PMI_Acceptor);
const double factor = pow (enorm, 2.0/3.0) * C * pow (Ni, lambda)
    * T0;
const double denom_ac_inv = B + factor / t;
const double dmu_ac_inv_dt =
    enorm * factor / (denom_ac_inv * denom_ac_inv * t * t);
const double damping = exp (-dist/l_crit);
dmuinvdt = damping * dmu_ac_inv_dt;
}

void Lombardi_EnormalMobility::
Compute_dmuinvdct (const double dist, const double pot,
                    const double enorm, const double n, const double p,
                    const double t, const double ct, double& dmuinvdct)
{ dmuinvdct = 0.0;
}

class Lombardi_e_EnormalMobility : public Lombardi_EnormalMobility {
public:
    Lombardi_e_EnormalMobility (const PMI_Environment& env,
                               const PMI_EnormalType type,
                               const PMI_AnisotropyType anisotype);

    ~Lombardi_e_EnormalMobility () {}

Lombardi_e_EnormalMobility:::
Lombardi_e_EnormalMobility (const PMI_Environment& env,
                           const PMI_EnormalType type,
                           const PMI_AnisotropyType anisotype) :
    Lombardi_EnormalMobility (env, type, anisotype)
{ // default values
B = InitParameter ("B_e", 4.750e7);
C = InitParameter ("C_e", 580.0);
lambda = InitParameter ("lambda_e", 0.125);
delta = InitParameter ("delta_e", 5.82e14);
eta = InitParameter ("eta_e", 5.82e30);
l_crit = InitParameter ("l_crit_e", 1.0e-6);
}

class Lombardi_h_EnormalMobility : public Lombardi_EnormalMobility {
public:
    Lombardi_h_EnormalMobility (const PMI_Environment& env,
                               const PMI_EnormalType type,
                               const PMI_AnisotropyType anisotype);

    ~Lombardi_h_EnormalMobility () {}

Lombardi_h_EnormalMobility:::
Lombardi_h_EnormalMobility (const PMI_Environment& env,
```

## Chapter 39: Physical Model Interface

### Semiconductor Band Structure

```
        const PMI_EnormalType type,
        const PMI_AnisotropyType anisotype) :
Lombardi_EnormalMobility (env, type, anisotype)
{ // default values
    B = InitParameter ("B_h", 9.925e6);
    C = InitParameter ("C_h", 2947.0);
    lambda = InitParameter ("lambda_h", 0.0317);
    delta = InitParameter ("delta_h", 2.0546e14);
    eta = InitParameter ("eta_h", 2.0546e30);
    l_crit = InitParameter ("l_crit_h", 1.0e-6);
}

extern "C"
PMI_EnormalMobility* new_PMI_Enormal_e_Mobility
(const PMI_Environment& env, const PMI_EnormalType type,
 const PMI_AnisotropyType anisotype)
{ return new Lombardi_e_EnormalMobility (env, type, anisotype);
}

extern "C"
PMI_EnormalMobility* new_PMI_Enormal_h_Mobility
(const PMI_Environment& env, const PMI_EnormalType type,
 const PMI_AnisotropyType anisotype)
{ return new Lombardi_h_EnormalMobility (env, type, anisotype);
}
```

---

## Semiconductor Band Structure

- [Apparent Band-Edge Shift](#)
  - [Band Gap](#)
  - [Bandgap Narrowing](#)
  - [Effective Mass](#)
  - [Electron Affinity](#)
- 

### Apparent Band-Edge Shift

The apparent band-edge shift  $\Lambda_{\text{PMI}}$  is a quantity similar to bandgap narrowing. In contrast to bandgap narrowing, the apparent band-edge shift can depend on the solution variables (electron and hole densities, lattice temperature, and electric field). Conversely, the apparent band-edge shift does not take effect in all situations where a real band-edge shift takes effect (this is why the band-edge shift is called *apparent*).

Implementationwise, the apparent band-edge shift is an extension of the density gradient model (see [Density Gradient Model on page 362](#)).

## Chapter 39: Physical Model Interface

### Semiconductor Band Structure

For the PMI model, this implies:

- Sentaurus Device applies the apparent band-edge shift  $\Lambda_{\text{PMI}}$  everywhere where it applies quantization corrections.
- By default, the apparent band-edge shift that Sentaurus Device computes is not equal to  $\Lambda_{\text{PMI}}$ , but contains contributions from quantization. To remove them, set  $\gamma = 0$  (see [Density Gradient Model on page 362](#)).
- Apart from a specification in the `Physics` section, it is necessary to specify additional equations in the `Solve` section (see [Using the Density Gradient Model on page 363](#)).

To select a model to compute  $\Lambda_{\text{PMI}}$ , specify its name using the `LocalModel` keyword (see [Table 299 on page 1676](#)). The same models for  $\Lambda_{\text{PMI}}$  can be used for the shift of the conduction and valence bands. A positive value of  $\Lambda_{\text{PMI}}$  means that the band shifts outwards, away from midgap (therefore, the band gap widens).

## Dependencies

The apparent band-edge shift  $\Lambda_{\text{PMI}}$  can depend on:

n	Electron density [cm <sup>-3</sup> ]
p	Hole density [cm <sup>-3</sup> ]
t	Lattice temperature [K]
f	Absolute value of the electric field [Vcm <sup>-1</sup> ]

The PMI model must compute the following values:

shift	Apparent band-edge shift $\Lambda_{\text{PMI}}$ [eV]
-------	--

In the case of the standard interface, the following derivatives must be computed as well:

dshiftdn	Derivative of $\Lambda_{\text{PMI}}$ with respect to n [eV cm <sup>3</sup> ]
dshiftdp	Derivative of $\Lambda_{\text{PMI}}$ with respect to p [eV cm <sup>3</sup> ]
dshiftdt	Derivative of $\Lambda_{\text{PMI}}$ with respect to t [eV K <sup>-1</sup> ]
dshiftdf	Derivative of $\Lambda_{\text{PMI}}$ with respect to f [eV cm V <sup>-1</sup> ]

## Chapter 39: Physical Model Interface

Semiconductor Band Structure

### Standard C++ Interface

The following base class is declared in the file `PMIModels.h`:

```
class PMI_ApparentBandEdgeShift : public PMI_Vertex_Interface {  
  
public:  
    PMI_ApparentBandEdgeShift (const PMI_Environment& env);  
    virtual ~PMI_ApparentBandEdgeShift ();  
  
    virtual void Compute_shift  
        (const double n, const double p,  
         const double t, const double f,  
         double& shift) = 0;  
  
    virtual void Compute_dshiftdn  
        (const double n, const double p,  
         const double t, const double f,  
         double& dshiftdn) = 0;  
  
    virtual void Compute_dshiftdp  
        (const double n, const double p,  
         const double t, const double f,  
         double& dshiftdp) = 0;  
  
    virtual void Compute_dshiftdt  
        (const double n, const double p,  
         const double t, const double f,  
         double& dshiftdt) = 0;  
  
    virtual void Compute_dshiftdf  
        (const double n, const double p,  
         const double t, const double f,  
         double& dshiftdf) = 0;  
};
```

The following virtual constructor must be implemented:

```
typedef PMI_ApparentBandEdgeShift* new_PMI_ApparentBandEdgeShift_func  
    (const PMI_Environment& env);  
extern "C" new_PMI_ApparentBandEdgeShift_func  
    new_PMI_ApparentBandEdgeShift;
```

### Simplified C++ Interface

The following base class is declared in the file `PMI.h`:

```
class PMI_ApparentBandEdgeShift_Base : public PMI_Vertex_Base {  
  
public:  
    class Input : public PMI_Vertex_Input_Base {  
    public:  
        pmi_float n; // electron density
```

## Chapter 39: Physical Model Interface

### Semiconductor Band Structure

```
    pmi_float p; // hole density
    pmi_float t; // lattice temperature
    pmi_float f; // absolute value of electric field
};

class Output {
public:
    pmi_float shift; // apparent band-edge shift
};

PMI_ApparentBandEdgeShift_Base (const PMI_Environment& env);
virtual ~PMI_ApparentBandEdgeShift_Base ();

virtual void compute (const Input& input, Output& output) = 0;
};
```

The prototype for the virtual constructor is given as:

```
typedef PMI_ApparentBandEdgeShift_Base*
new_PMI_ApparentBandEdgeShift_Base_func (const PMI_Environment&
env);
extern "C" new_PMI_ApparentBandEdgeShift_Base_func
new_PMI_ApparentBandEdgeShift_Base;
```

---

## Band Gap

Sentaurus Device provides a PMI to compute the energy band gap  $E_g$  in a semiconductor. It can be specified in the `Physics` section of the command file. For example:

```
Physics {
    EffectiveIntrinsicDensity (
        BandGap (pmi_model_name)
    )
}
```

The default bandgap model in Sentaurus Device is selected explicitly by the keyword `Default`:

```
Physics {
    EffectiveIntrinsicDensity (
        BandGap (Default)
    )
}
```

## Dependencies

The band gap  $E_g$  can depend on:

$t$  Lattice temperature [K]

## Chapter 39: Physical Model Interface

### Semiconductor Band Structure

The PMI model must compute the following results:

$\text{bg}$  Band gap  $E_g$  [eV]

In the case of the standard interface, the following derivatives must be computed as well:

$\text{dbgdt}$  Derivative of  $\text{bg}$  with respect to  $t$  [eV K $^{-1}$ ]

## Standard C++ Interface

The following base class is declared in the file `PMIModels.h`:

```
class PMI_BandGap : public PMI_Vertex_Interface {  
  
public:  
    PMI_BandGap (const PMI_Environment& env);  
    virtual ~PMI_BandGap ();  
  
    virtual void Compute_bg  
        (const double t, double& bg) = 0;  
  
    virtual void Compute_dbgdt  
        (const double t, double& dbgdt) = 0;  
};
```

The prototype for the virtual constructor is given as:

```
typedef PMI_BandGap* new_PMI_BandGap_func (const PMI_Environment& env);  
extern "C" new_PMI_BandGap_func new_PMI_BandGap;
```

## Simplified C++ Interface

The following base class is declared in the file `PMI.h`:

```
class PMI_BandGap_Base : public PMI_Vertex_Base {  
  
public:  
    class Input : public PMI_Vertex_Input_Base {  
    public:  
        pmi_float t;      // lattice temperature  
    };  
  
    class Output {  
    public:  
        pmi_float bg;    // band gap  
    };  
  
    PMI_BandGap_Base (const PMI_Environment& env);  
    virtual ~PMI_BandGap_Base ();
```

## Chapter 39: Physical Model Interface

### Semiconductor Band Structure

```
    virtual void compute (const Input& input, Output& output) = 0;  
};
```

The prototype for the virtual constructor is given as:

```
typedef PMI_BandGap_Base* new_PMI_BandGap_Base_func  
    (const PMI_Environment& env);  
extern "C" new_PMI_BandGap_Base_func new_PMI_BandGap_Base;
```

## Example: Default Temperature Dependence

Sentaurus Device uses the following default bandgap model:

$$E_g(t) = E_g(0) - \frac{\alpha t^2}{t + \beta} \quad (1342)$$

$E_g(0)$  denotes the band gap at 0 K.

```
#include "PMIModels.h"  
  
class Default_BandGap : public PMI_BandGap {  
  
private:  
    double Eg0, alpha, beta;  
  
public:  
    Default_BandGap (const PMI_Environment& env);  
  
    ~Default_BandGap ();  
  
    void Compute_bg (const double t, double& bg);  
  
    void Compute_dbgdt (const double t, double& dbgdt);  
};  
  
Default_BandGap::  
Default_BandGap (const PMI_Environment& env) :  
    PMI_BandGap (env)  
{  
    Eg0 = InitParameter ("Eg0", 1.16964);  
    alpha = InitParameter ("alpha", 4.73e-4);  
    beta = InitParameter ("beta", 636);  
}  
  
Default_BandGap::  
~Default_BandGap ()  
{  
}  
  
void Default_BandGap::  
Compute_bg (const double t, double& bg)  
{  
    bg = Eg0 - alpha * t * t / (t + beta);  
}
```

## Chapter 39: Physical Model Interface

### Semiconductor Band Structure

```
}
```

```
void Default_BandGap::
```

```
Compute_dbgdt (const double t, double& dbgdt)
```

```
{ dbgdt = - alpha * t * (t + 2.0 * beta) / ((t + beta) * (t + beta));
```

```
}
```

```
extern "C"
```

```
PMI_BandGap* new_PMI_BandGap
```

```
(const PMI_Environment& env)
```

```
{ return new Default_BandGap (env);
```

```
}
```

---

## Bandgap Narrowing

Sentaurus Device provides a PMI to compute bandgap narrowing (see [Band Gap and Electron Affinity on page 305](#)). A user model is activated with the keyword `EffectiveIntrinsicDensity` in the `Physics` section of the command file:

```
Physics {
```

```
    EffectiveIntrinsicDensity (pmi_model_name)
```

```
}
```

## Dependencies

A PMI bandgap narrowing model has no explicit dependencies. However, it can depend on doping concentrations through the runtime support.

The PMI model must compute:

`bgn`      Bandgap narrowing  $\Delta E_g^0$  [eV]

In most cases, it is not necessary to compute the derivatives with respect to the dopant concentrations. However, to model dopant fluctuations (see [Chapter 23 on page 781](#)), in the standard interface the PMI model must override the functions that compute the following values:

`dbgndNa`      Derivative of  $\Delta E_g^0$  with respect to the acceptor concentration [cm<sup>3</sup>eV]

`dbgndNd`      Derivative of  $\Delta E_g^0$  with respect to the donor concentration [cm<sup>3</sup>eV]

## Chapter 39: Physical Model Interface

### Semiconductor Band Structure

## Standard C++ Interface

The following base class is declared in the file `PMIModels.h`:

```
class PMI_BandGapNarrowing : public PMI_Vertex_Interface {
public:
    PMI_BandGapNarrowing (const PMI_Environment& env);
    virtual ~PMI_BandGapNarrowing ();
    virtual void Compute_bgn (double& bgn) = 0;
    virtual void Compute_dbgndNa (double& dbgndNa);
    virtual void Compute_dbgndNd (double& dbgndNd);
};
```

The following virtual constructor must be implemented:

```
typedef PMI_BandGapNarrowing* new_PMI_BandGapNarrowing_func
(const PMI_Environment& env);
extern "C" new_PMI_BandGapNarrowing_func new_PMI_BandGapNarrowing;
```

## Simplified C++ Interface

The following base class is declared in the file `PMI.h`:

```
class PMI_BandGapNarrowing_Base : public PMI_Vertex_Base {
public:
    class Input : public PMI_Vertex_Input_Base {
public:
    pmi_float acceptor; // total acceptor concentration
    pmi_float donor; // total donor concentration
};

    class Output {
public:
    pmi_float bgn; // bandgap narrowing
};

    PMI_BandGapNarrowing_Base (const PMI_Environment& env);
    virtual ~PMI_BandGapNarrowing_Base ();

    virtual void compute (const Input& input, Output& output) = 0;
};
```

The prototype for the virtual constructor is given as:

```
typedef PMI_BandGapNarrowing_Base* new_PMI_BandGapNarrowing_Base_func
(const PMI_Environment& env);
extern "C" new_PMI_BandGapNarrowing_Base_func
new_PMI_BandGapNarrowing_Base;
```

## Example: Bennett–Wilson Bandgap Narrowing

The default bandgap narrowing model in Sentaurus Device (Bennett–Wilson) is given by:

$$\Delta E_g^0 = \begin{cases} \overset{\circ}{\underset{\circ}{E}}_{\text{ref}} \left[ \ln \frac{N_{\text{tot}}}{N_{\text{ref}}} \right]^2, & N_{\text{tot}} > N_{\text{ref}}, \\ 0, & N_{\text{tot}} \leq N_{\text{ref}}. \end{cases} \quad (1343)$$

See [Band Gap and Electron Affinity on page 305](#).

This model can be implemented as a PMI model as follows:

```
#include "PMIModels.h"

class Bennett_BandGapNarrowing : public PMI_BandGapNarrowing {

private:
    double Ebgn, Nref;

public:
    Bennett_BandGapNarrowing (const PMI_Environment& env);

    ~Bennett_BandGapNarrowing ();

    void Compute_bgn (double& bgn);
};

Bennett_BandGapNarrowing::Bennett_BandGapNarrowing (const PMI_Environment& env) :
    PMI_BandGapNarrowing (env)
{
    Ebgn = InitParameter ("Ebgn", 6.84e-3);
    Nref = InitParameter ("Nref", 3.162e18);
}

Bennett_BandGapNarrowing::~Bennett_BandGapNarrowing ()
{
}

void Bennett_BandGapNarrowing::Compute_bgn (double& bgn)
{
    const double Na = ReadDoping (PMI_Acceptor);
    const double Nd = ReadDoping (PMI_Donor);
    const double Ni = Na + Nd;
    if (Ni > Nref) {
        const double tmp = log (Ni / Nref);
        bgn = Ebgn * tmp * tmp;
    } else {
        bgn = 0.0;
    }
}
```

## Chapter 39: Physical Model Interface

### Semiconductor Band Structure

```
extern "C"
PMI_BandGapNarrowing* new_PMI_BandGapNarrowing
    (const PMI_Environment& env)
{ return new Bennett_BandGapNarrowing (env);
}
```

---

## Effective Mass

Sentaurus Device provides a PMI to compute the effective mass of electrons and holes. The effective mass is always expressed as a multiple of the electron mass in vacuum. The name of the PMI model must appear in the `Physics` section of the command file:

```
Physics { EffectiveMass (pmi_model_name) }
```

## Dependencies

The relative effective mass can depend on the following variables:

`t` Lattice temperature [K]  
`bg` Band gap [eV]

The PMI model must compute the following results:

`m` Relative effective mass (1)

In the case of the standard interface, the following derivatives must be computed as well:

`dmdt` Derivative of `m` with respect to `t` [ $K^{-1}$ ]  
`dmdbg` Derivative of `m` with respect to `bg` [ $eV^{-1}$ ]

## Standard C++ Interface

The following base class is declared in the file `PMIModels.h`:

```
class PMI_EffectiveMass : public PMI_Vertex_Interface {

public:
    PMI_EffectiveMass (const PMI_Environment& env);
    virtual ~PMI_EffectiveMass ();

    virtual void Compute_m
        (const double t, const double bg, double& m) = 0;

    virtual void Compute_dmdt
```

## Chapter 39: Physical Model Interface

### Semiconductor Band Structure

```
(const double t, const double bg, double& dmdbg) = 0;  
  
virtual void Compute_dmdbg  
    (const double t, const double bg, double& dmdbg) = 0;  
};
```

Two virtual constructors are necessary to compute the effective mass of electrons and holes:

```
typedef PMI_EffectiveMass* new_PMI_EffectiveMass_func  
    (const PMI_Environment& env);  
extern "C" new_PMI_EffectiveMass_func new_PMI_e_EffectiveMass;  
extern "C" new_PMI_EffectiveMass_func new_PMI_h_EffectiveMass;
```

## Simplified C++ Interface

The following base class is declared in the file `PMI.h`:

```
class PMI_EffectiveMass_Base : public PMI_Vertex_Base {  
  
public:  
    class Input : public PMI_Vertex_Input_Base {  
public:  
    pmi_float t; // lattice temperature  
    pmi_float bg; // band gap  
};  
  
    class Output {  
public:  
    pmi_float m; // effective mass  
};  
  
    PMI_EffectiveMass_Base (const PMI_Environment& env);  
    virtual ~PMI_EffectiveMass_Base ();  
  
    virtual void compute (const Input& input, Output& output) = 0;  
};
```

The prototype for the virtual constructor is given as:

```
typedef PMI_EffectiveMass_Base* new_PMI_EffectiveMass_Base_func  
    (const PMI_Environment& env);  
extern "C" new_PMI_EffectiveMass_Base_func  
    new_PMI_e_EffectiveMass_Base;  
extern "C" new_PMI_EffectiveMass_Base_func  
    new_PMI_h_EffectiveMass_Base;
```

## Chapter 39: Physical Model Interface

### Semiconductor Band Structure

## Example: Linear Effective Mass

A simple, linear effective mass model is given by:

$$m = m_{300} + \frac{dm}{dt}(t - 300) \quad (1344)$$

$m_{300}$  denotes the mass at 300 K. It can be implemented as follows:

```
#include "PMIModels.h"

class Linear_EffectiveMass : public PMI_EffectiveMass {

protected:
    double mass_300, dmass_dt;

public:
    Linear_EffectiveMass (const PMI_Environment& env);

    ~Linear_EffectiveMass ();

    void Compute_m (const double t, const double bg, double& m);

    void Compute_dmdt (const double t, const double bg, double& dmdt);

    void Compute_dmdbg (const double t, const double bg, double& dmdbg);
};

Linear_EffectiveMass::
Linear_EffectiveMass (const PMI_Environment& env) :
    PMI_EffectiveMass (env)
{
}

Linear_EffectiveMass::
~Linear_EffectiveMass ()
{
}

void Linear_EffectiveMass::
Compute_m (const double t, const double bg, double& m)
{
    m = mass_300 + dmass_dt * (t - 300.0);
}

void Linear_EffectiveMass::
Compute_dmdt (const double t, const double bg, double& dmdt)
{
    dmdt = dmass_dt;
}

void Linear_EffectiveMass::
Compute_dmdbg (const double t, const double bg, double& dmdbg)
{
    dmdbg = 0.0;
}

class Linear_e_EffectiveMass : public Linear_EffectiveMass {
```

## Chapter 39: Physical Model Interface

### Semiconductor Band Structure

```
public:  
    Linear_e_EffectiveMass (const PMI_Environment& env);  
  
    ~Linear_e_EffectiveMass () {}  
  
};  
  
Linear_e_EffectiveMass::  
Linear_e_EffectiveMass (const PMI_Environment& env) :  
    Linear_EffectiveMass (env)  
{ mass_300 = InitParameter ("mass_e_300", 1.09);  
    dmass_dt = InitParameter ("dmass_e_dt", 1.6e-4);  
}  
  
class Linear_h_EffectiveMass : public Linear_EffectiveMass {  
  
public:  
    Linear_h_EffectiveMass (const PMI_Environment& env);  
  
    ~Linear_h_EffectiveMass () {}  
  
};  
  
Linear_h_EffectiveMass::  
Linear_h_EffectiveMass (const PMI_Environment& env) :  
    Linear_EffectiveMass (env)  
{ mass_300 = InitParameter ("mass_h_300", 1.15);  
    dmass_dt = InitParameter ("dmass_h_dt", 9.2e-4);  
}  
  
extern "C"  
PMI_EffectiveMass* new_PMI_e_EffectiveMass  
    (const PMI_Environment& env)  
{ return new Linear_e_EffectiveMass (env);  
}  
  
extern "C"  
PMI_EffectiveMass* new_PMI_h_EffectiveMass  
    (const PMI_Environment& env)  
{ return new Linear_h_EffectiveMass (env);  
}
```

---

## Electron Affinity

The electron affinity  $\chi$ , that is, the energy separation between the conduction band and vacuum level, can be specified by using a PMI. The syntax in the command file is:

```
Physics { Affinity (pmi_model_name) }
```

## Chapter 39: Physical Model Interface

### Semiconductor Band Structure

The default affinity model in Sentaurus Device can be selected explicitly by the keyword Default:

```
Physics { Affinity (Default)}
```

## Dependencies

The electron affinity  $\chi$  can depend on:

$t$  Lattice temperature [K]

The PMI model must compute:

```
affinity    Electron affinity  $\chi$  [eV]
```

In the case of the standard interface, the following derivative must be computed as well:

```
affinitydt  Derivative of affinity with respect to  $t$  [eVK $^{-1}$ ]
```

## Standard C++ Interface

The following base class is declared in the file `PMIModels.h`:

```
class PMI_Affinity : public PMI_Vertex_Interface {

public:
    PMI_Affinity (const PMI_Environment& env);
    virtual ~PMI_Affinity () ;

    virtual void Compute_affinity
        (const double t, double& affinity) = 0;

    virtual void Compute_daffinitydt
        (const double t, double& daffinitydt) = 0;
};
```

The prototype for the virtual constructor is:

```
typedef PMI_Affinity* new_PMI_Affinity_func
    (const PMI_Environment& env);
extern "C" new_PMI_Affinity_func new_PMI_Affinity;
```

## Simplified C++ Interface

The following base class is declared in the file `PMI.h`:

```
class PMI_Affinity_Base : public PMI_Vertex_Base {

public:
    class Input : public PMI_Vertex_Input_Base {
public:
    pmi_float t; // lattice temperature
};

    class Output {
public:
    pmi_float affinity; // electron affinity
};

    PMI_Affinity_Base (const PMI_Environment& env);
    virtual ~PMI_Affinity_Base ();

    virtual void compute (const Input& input, Output& output) = 0;
};
```

The prototype for the virtual constructor is given as:

```
typedef PMI_Affinity_Base* new_PMI_Affinity_Base_func
    (const PMI_Environment& env);
extern "C" new_PMI_Affinity_Base_func new_PMI_Affinity_Base;
```

## Example: Default Temperature Dependence

By default, Sentaurus Device uses this formula to compute  $\chi$ :

$$\chi(t) = \chi(0) + 0.5 \frac{\alpha t^2}{t + \beta} \quad (1345)$$

$\chi(0)$  denotes the affinity at 0 K.

```
#include "PMIModels.h"

class Default_Affinity : public PMI_Affinity {

private:
    double Affinity0, alpha, beta;

public:
    Default_Affinity (const PMI_Environment& env);

    ~Default_Affinity ();

    void Compute_affinity (const double t, double& affinity);

    void Compute_daffinitydt (const double t, double& daffinitydt);
```

## Chapter 39: Physical Model Interface

### Phase and State Transitions

```
};

Default_Affinity::
Default_Affinity (const PMI_Environment& env) :
    PMI_Affinity (env)
{
Affinity0 = InitParameter ("Affinity0", 4.05);
    alpha = InitParameter ("alpha", 4.73e-4);
    beta = InitParameter ("beta", 636);
}

Default_Affinity::
~Default_Affinity ()
{
}

void Default_Affinity::
Compute_affinity (const double t, double& affinity)
{ affinity = Affinity0 + 0.5 * alpha * t * t / (t + beta);
}

void Default_Affinity::
Compute_daffinitydt (const double t, double& daffinitydt)
{ daffinitydt = 0.5 * alpha * t * (t + 2.0 * beta) /
    ((t + beta) * (t + beta));
}

extern "C"
PMI_Affinity* new_PMI_Affinity
    (const PMI_Environment& env)
{ return new Default_Affinity (env)
}
```

---

## Phase and State Transitions

This section presents the following PMIs:

- Multistate Configuration–Dependent Apparent Band-Edge Shift
- Multistate Configuration–Dependent Bulk Mobility
- Multistate Configuration–Dependent Heat Capacity
- Multistate Configuration–Dependent Thermal Conductivity

---

### Multistate Configuration–Dependent Apparent Band-Edge Shift

The multistate configuration (MSC)–dependent, apparent band-edge shift model is a variant of the apparent band-edge shift model (see [Apparent Band-Edge Shift on page 1308](#)). Besides dependencies on the solution variables, electron and hole densities, lattice

## Chapter 39: Physical Model Interface

### Phase and State Transitions

temperature, and carrier temperatures, it allows dependencies on all state occupation rates of an arbitrary reference MSC defined by `MSConfig`. The remarks made for the apparent band-edge shift in connection with the density gradient model are valid here as well.

The model can be selected as arguments of the keywords `eBandEdgeShift`, `hBandEdgeShift`, and `BandEdgeShift` (see [Apparent Band-Edge Shift on page 595](#)) in the `MSConfig` specification. The optional model index parameter allows you to implement, in the same model, several variants that can be accessed from the command file.

## Dependencies

The MSC apparent band-edge shift  $\Lambda_{\text{PMI}}$  can depend on:

- n Electron density [ $\text{cm}^{-3}$ ]
- p Hole density [ $\text{cm}^{-3}$ ]
- t Lattice temperature [K]
- ct Carrier temperature [K]
- s Vector of state occupation rates of the reference MSC [1]

The PMI model must compute the following values if the dependency is used:

`shift` Apparent band-edge shift  $\Lambda_{\text{PMI}}$  [eV]

In the case of the standard interface, the following derivatives must be computed as well:

`dshiftdn` Derivative of  $\Lambda_{\text{PMI}}$  with respect to n [ $\text{eV cm}^3$ ]

`dshiftdp` Derivative of  $\Lambda_{\text{PMI}}$  with respect to p [ $\text{eV cm}^3$ ]

`dshiftdt` Derivative of  $\Lambda_{\text{PMI}}$  with respect to t [ $\text{eVK}^{-1}$ ]

`dshiftdf` Derivative of  $\Lambda_{\text{PMI}}$  with respect to ct [ $\text{eVK}^{-1}$ ]

`dshiftds` Derivative of  $\Lambda_{\text{PMI}}$  with respect to s [eV]

## Additional Functionality

The PMI model provides additional functionality.

### Using Dependencies

You can use the function `set_dependency_used` to switch on or off the dependencies of this model explicitly (the default is on). For used dependencies, the function computing the corresponding derivative must be provided; for unused dependencies, the functions are not called.

### Updating Actual Status

Before calling the computation functions (`compute_val` and `compute_dval_dx`), the simulator passes the actual values of the dependencies to the model using `set_actual_status`. The model parameters are updated by `init_parameter` before the actual status is updated.

## Standard C++ Interface

The following base class (here, only an extract) is declared in the file `PMIModels.h`:

```
class PMI_MSC_ApparentBandEdgeShift: public PMI_MSC_Vertex_Interface {  
  
public:  
    enum e_var { var_n, var_p, var_T, var_eT, var_hT, var_s,  
                var_undefined };  
  
    class input_data {  
    public:  
        input_data ();  
        ~input_data ();  
        double& val ( e_var var, size_t ind );  
        double val ( e_var var, size_t ind ) const;  
    };  
  
    public:  
        PMI_MSC_ApparentBandEdgeShift (const PMI_Environment& env,  
                                         const std::string& msconfig_name, int model_index = 0);  
        virtual ~PMI_MSC_ApparentBandEdgeShift ();  
  
        // get names of MSConfig and its states  
        const std::string& msconfig_name () const;  
        size_t nb_states () const;  
        const std::string& state ( size_t index ) const;  
  
        virtual void set_actual_status (  
            const PMI_MSC_ApparentBandEdgeShift::input_data& id );  
  
        // compute value and derivatives  
        virtual void compute_val ( double& val );
```

## Chapter 39: Physical Model Interface

### Phase and State Transitions

```
virtual void compute_dval_dn ( double& val );
virtual void compute_dval_dp ( double& val );
virtual void compute_dval_dT ( double& val );
virtual void compute_dval_deT ( double& val );
virtual void compute_dval_dHT ( double& val );
virtual void compute_dval_ds ( std::vector<double>& val );

// support ramping of parameters
virtual void init_parameter ();

// handle dependencies
void set_dependency_used (
    PMI_MSC_ApparentBandEdgeShift::e_var var, bool flag );
bool dependency_used ( PMI_MSC_ApparentBandEdgeShift::e_var var )
    const;
};
```

The following virtual constructor must be implemented:

```
typedef PMI_MSC_ApparentBandEdgeShift*
    new_PMI_MSC_ApparentBandEdgeShift_func
    (const PMI_Environment& env,
     const std::string& msconfig_name,
     int model_index);
extern "C" new_PMI_ApparentBandEdgeShift_func
new_PMI_MSC_ApparentBandEdgeShift;
```

## Simplified C++ Interface

The following base class is declared in the file `PMI.h`:

```
class PMI_MSC_ApparentBandEdgeShift_Base : public
    PMI_MSC_Vertex_Base {

public:
    class Input : public PMI_Vertex_Input_Base {
public:
    pmi_float n; // electron density
    pmi_float p; // hole density
    pmi_float T; // lattice temperature
    pmi_float eT; // electron temperature
    pmi_float hT; // hole temperature
    std::vector<pmi_float> s; // phase fraction
};

    class Output {
public:
    pmi_float val; // apparent band-edge shift
};

    PMI_MSC_ApparentBandEdgeShift_Base (const PMI_Environment& env,
                                         const std::string&
                                         msconfig_name,
```

## Chapter 39: Physical Model Interface

### Phase and State Transitions

```
        const int model_index);  
virtual ~PMI_MSC_ApparentBandEdgeShift_Base ();  
  
virtual void compute (const Input& input, Output& output) = 0;  
};
```

The prototype for the virtual constructor is given as:

```
typedef PMI_MSC_ApparentBandEdgeShift_Base*  
new_PMI_MSC_ApparentBandEdgeShift_Base_func  
(const PMI_Environment& env, const std::string& msconfig_name,  
const int model_index);  
extern "C" new_PMI_MSC_ApparentBandEdgeShift_Base_func  
new_PMI_MSC_ApparentBandEdgeShift_Base;
```

---

## Multistate Configuration–Dependent Bulk Mobility

This PMI allows you to implement multistate configuration (MSC)–dependent bulk mobility models.

### Command File

To activate a PMI of this type, as an option to `eMobility`, `hMobility`, or `Mobility` in the `Physics` section, specify:

```
DopingDependence(  
    PMIModel (  
        Name = <string>  
        MSCConfig = <string>  
        Index = <int>  
        String = <string>  
    )  
)
```

The options of `PMIModel` are described in [Command File on page 1332](#).

### Dependencies

The mobility can depend on the variables:

- n      Electron density [cm<sup>-3</sup>]
- p      Hole density [cm<sup>-3</sup>]
- T      Lattice temperature [K]
- eT     Electron temperature [K]

## Chapter 39: Physical Model Interface

### Phase and State Transitions

hT Hole temperature [K]

s Multistate configuration occupation probabilities [1]

The model must compute the following quantities:

val Mobility  $\mu_{\text{dop}}$  [ $\text{cm}^2 \text{V}^{-1} \text{s}^{-1}$ ]

In the case of the standard interface, the following derivatives must be computed as well:

dval\_dn Derivative with respect to electron density [ $\text{cm}^5 \text{V}^{-1} \text{s}^{-1}$ ]

dval\_dp Derivative with respect to hole density [ $\text{cm}^5 \text{V}^{-1} \text{s}^{-1}$ ]

dval\_dT Derivative with respect to lattice temperature [ $\text{cm}^2 \text{V}^{-1} \text{s}^{-1} \text{K}^{-1}$ ]

dval\_deT Derivative with respect to electron temperature [ $\text{cm}^2 \text{V}^{-1} \text{s}^{-1} \text{K}^{-1}$ ]

dval\_dhT Derivative with respect to hole temperature [ $\text{cm}^2 \text{V}^{-1} \text{s}^{-1} \text{K}^{-1}$ ]

dval\_ds Derivative with respect to multistate configuration occupation probabilities [ $\text{cm}^2 \text{V}^{-1} \text{s}^{-1}$ ]

## Standard C++ Interface

The PMI offers a base class that presents the following interface:

```
class PMI_MSC_Mobility : public PMI_MSC_Vertex_Interface
{
public:
    PMI_MSC_Mobility (const PMI_Environment& env,
                      const std::string& msconfig_name,
                      const int model_index,
                      const std::string& model_string,
                      const PMI_AnisotropyType aniso);
    // otherwise, see Standard C++ Interface on page 1892
};
```

Apart from the name of the base class and the constructor, the explanations in [Standard C++ Interface on page 1334](#) apply here as well.

The following virtual constructor must be implemented:

```
typedef PMI_MSC_Mobility* new_PMI_MSC_Mobility_func
(const PMI_Environment& env, const std::string& msconfig_name,
```

## Chapter 39: Physical Model Interface

### Phase and State Transitions

```
int model_index, const std::string& model_string,
const PMI_AnisotropyType anisotype);
extern "C" new_PMI_MSC_Mobility_func new_PMI_MSC_Mobility;
```

## Simplified C++ Interface

The following base class is declared in the file `PMI.h`:

```
class PMI_MSC_Mobility_Base : public PMI_MSC_Vertex_Base {

public:
    class Input : public PMI_Vertex_Input_Base {
public:
    const pmi_float& n () const; // electron density
    const pmi_float& p () const; // hole density
    const pmi_float& T () const; // lattice temperature
    const pmi_float& eT () const; // electron temperature
    const pmi_float& hT () const; // hole temperature
    const pmi_float& s (size_t ind) const; // phase fraction
};

    class Output {
public:
    pmi_float& val (); // mobility
};

    PMI_MSC_Mobility_Base (const PMI_Environment& env,
                           const std::string& msconfig_name,
                           const int model_index,
                           const std::string& model_string,
                           const PMI_AnisotropyType anisotype);
    virtual ~PMI_MSC_Mobility_Base ();

    PMI_AnisotropyType AnisotropyType () const;

    virtual void compute (const Input& input, Output& output) = 0;
};
```

The prototype for the virtual constructor is given as:

```
typedef PMI_MSC_Mobility_Base* new_PMI_MSC_Mobility_Base_func
(const PMI_Environment& env, const std::string& msconfig_name,
const int model_index, const std::string& model_string,
const PMI_AnisotropyType anisotype);
extern "C" new_PMI_MSC_Mobility_Base_func new_PMI_MSC_Mobility_Base;
```

---

## Multistate Configuration-Dependent Heat Capacity

This PMI computes the lattice heat capacity and allows it to depend on a multistate configuration (see [Chapter 18 on page 584](#)).

## Chapter 39: Physical Model Interface

Phase and State Transitions

### Command File

To activate a PMI of this type, in the `Physics` section, specify:

```
HeatCapacity(  
    PMIModel (  
        Name = <string>  
        MSCConfig = <string>  
        Index = <int>  
        String = <string>  
    )  
)
```

The options of `PMIModel` are described in [Command File on page 1332](#).

### Dependencies

The heat capacity can depend on the variables:

n	Electron density [ $\text{cm}^{-3}$ ]
p	Hole density [ $\text{cm}^{-3}$ ]
T	Lattice temperature [K]
eT	Electron temperature [K]
hT	Hole temperature [K]
s	Multistate configuration occupation probabilities [1]

The model must compute the following quantities:

val	Heat capacity [ $\text{Jcm}^{-3}\text{K}^{-1}$ ]
-----	--

In the case of the standard interface, the following derivatives must be computed as well:

dval_dn	Derivative with respect to electron density [ $\text{JK}^{-1}$ ]
---------	--

dval_dp	Derivative with respect to hole density [ $\text{JK}^{-1}$ ]
---------	--

dval_dT	Derivative with respect to lattice temperature [ $\text{Jcm}^{-3}\text{K}^{-2}$ ]
---------	---

## Chapter 39: Physical Model Interface

### Phase and State Transitions

dval_deT	Derivative with respect to electron temperature [ $\text{Jcm}^{-3}\text{K}^{-2}$ ]
dval_dhT	Derivative with respect to hole temperature [ $\text{Jcm}^{-3}\text{K}^{-2}$ ]
dval_ds	Derivative with respect to multistate configuration occupation probabilities [ $\text{Jcm}^{-3}\text{K}^{-1}$ ]

## Standard C++ Interface

The PMI offers a base class that presents the following interface:

```
class PMI_MSC_HeatCapacity : public PMI_MSC_Vertex_Interface
{
public:
    PMI_MSC_HeatCapacity (const PMI_Environment& env,
                          const std::string& msconfig_name,
                          const int model_index,
                          const std::string& model_string);
    // otherwise, see Standard C++ Interface on page 1892
};
```

Apart from the base class name and the constructor, the explanations in [Standard C++ Interface on page 1334](#) apply here as well.

The following virtual constructor must be implemented:

```
typedef PMI_MSC_HeatCapacity* new_PMI_MSC_HeatCapacity_func
    (const PMI_Environment& env, const std::string& msconfig_name,
     int model_index, const std::string& model_string);
extern "C" new_PMI_MSC_HeatCapacity_func new_PMI_MSC_HeatCapacity;
```

## Simplified C++ Interface

The following base class is declared in the file `PMI.h`:

```
class PMI_MSC_HeatCapacity_Base : public PMI_MSC_Vertex_Base {

public:
    class Input : public PMI_Vertex_Input_Base {
public:
    const pmi_float& n () const; // electron density
    const pmi_float& p () const; // hole density
    const pmi_float& T () const; // lattice temperature
    const pmi_float& eT () const; // electron temperature
    const pmi_float& hT () const; // hole temperature
    const pmi_float& s (size_t ind) const; // phase fraction
};
```

## Chapter 39: Physical Model Interface

### Phase and State Transitions

```
class Output {
public:
    Output (NS_PMI_MSC::odata* odata);

    pmi_float& val () ; // heat capacity
};

PMI_MSC_HeatCapacity_Base (const PMI_Environment& env,
                           const std::string& msconfig_name,
                           const int model_index,
                           const std::string& model_string);
virtual ~PMI_MSC_HeatCapacity_Base ();

virtual void compute (const Input& input, Output& output) = 0;
};
```

The prototype for the virtual constructor is given as:

```
typedef PMI_MSC_Mobility_Base* new_PMI_MSC_Mobility_Base_func
(const PMI_Environment& env, const std::string& msconfig_name,
const int model_index, const std::string& model_string,
const PMI_AnisotropyType anisotype);
extern "C" new_PMI_MSC_HeatCapacity_Base_func
new_PMI_MSC_HeatCapacity_Base;
```

---

## Multistate Configuration–Dependent Thermal Conductivity

This PMI provides access to the lattice thermal conductivity  $\kappa$  in [Equation 71 on page 248](#) and allows it to depend on a multistate configuration (see [Chapter 18 on page 584](#)).

### Command File

To activate a PMI of this type, in the `Physics` section, specify:

```
ThermalConductivity(
    PMIModel (
        Name = <string>
        MSConfig = <string>
        Index = <int>
        String = <string>
    )
)
```

Here, `Name` is the name of the PMI model; its specification is mandatory. `MSConfig` selects the name of the multistate configuration. `MSConfig` defaults to an empty string, which means the model does not depend on any multistate configuration. `Index` and `String` are optional; they determine the arguments `model_index` and `model_string` that are passed to the virtual constructor (see below); the interpretation of those arguments is up to the PMI model. `Index` defaults to zero, and `String` defaults to the empty string.

## Chapter 39: Physical Model Interface

### Phase and State Transitions

## Dependencies

The thermal conductivity can depend on the variables:

- n Electron density [ $\text{cm}^{-3}$ ]
- p Hole density [ $\text{cm}^{-3}$ ]
- T Lattice temperature [K]
- eT Electron temperature [K]
- hT Hole temperature [K]
- s Multistate configuration occupation probabilities [1]

The model must compute the following quantities:

- val Thermal conductivity [ $\text{Wcm}^{-1}\text{K}^{-1}$ ]

In the case of the standard interface, the following derivatives must be computed as well:

- dval\_dn Derivative with respect to electron density [ $\text{Wcm}^2\text{K}^{-1}$ ]
- dval\_dp Derivative with respect to hole density [ $\text{Wcm}^2\text{K}^{-1}$ ]
- dval\_dT Derivative with respect to lattice temperature [ $\text{Wcm}^{-1}\text{K}^{-2}$ ]
- dval\_deT Derivative with respect to electron temperature [ $\text{Wcm}^{-1}\text{K}^{-2}$ ]
- dval\_dhT Derivative with respect to hole temperature [ $\text{Wcm}^{-1}\text{K}^{-2}$ ]
- dval\_ds Derivative with respect to multistate configuration occupation probabilities [ $\text{Wcm}^{-1}\text{K}^{-1}$ ]

## Chapter 39: Physical Model Interface

Phase and State Transitions

### Standard C++ Interface

The PMI offers a base class that presents the following interface:

```
class PMI_MSC_ThermalConductivity : public PMI_MSC_Vertex_Interface
{
public:
    class idata {
public:
    double n () const;
    double p () const;
    double T () const;
    double eT () const;
    double hT () const;
    double s (size_t ind) const;
};

    class odata {
public:
    double& val ();
    double& dval_dn ();
    double& dval_dp ();
    double& dval_dT ();
    double& dval_deT ();
    double& dval_dhT ();
    double& dval_ds ( size_t ind );
};

PMI_MSC_ThermalConductivity(const PMI_Environment& env,
    const std::string& msconfig_name,
    const int model_index,
    const std::string& model_string,
    const PMI_AnisotropyType anisotype);
virtual ~PMI_MSC_ThermalConductivity ();

PMI_AnisotropyType AnisotropyType () const;

virtual void compute
    (const idata* id,
     odata* od ) = 0;
};
```

The `Compute` function receives its input from `id`. It returns the results using `od` by assignment using the member functions of `odata`. The PMI framework initializes the values of the derivatives to zero, so you do not have to do anything for derivatives with respect to the variables on which your model does not depend.

The following virtual constructor must be implemented:

```
typedef PMI_MSC_ThermalConductivity*
    new_PMI_MSC_ThermalConductivity_func
    (const PMI_Environment& env, const std::string& msconfig_name,
    int model_index, const std::string& model_string,
```

## Chapter 39: Physical Model Interface

### Phase and State Transitions

```
    const PMI_AnisotropyType anisotype);
extern "C" new_PMI_MSC_ThermalConductivity_func
new_PMI_MSC_ThermalConductivity;
```

## Simplified C++ Interface

The following base class is declared in the file `PMI.h`:

```
class PMI_MSC_ThermalConductivity_Base : public PMI_MSC_Vertex_Base {

public:
    class Input : public PMI_Vertex_Input_Base {
public:
    const pmi_float& n () const; // electron density
    const pmi_float& p () const; // hole density
    const pmi_float& T () const; // lattice temperature
    const pmi_float& eT () const; // electron temperature
    const pmi_float& hT () const; // hole temperature
    const pmi_float& s (size_t ind) const; // phase fraction
};

    class Output {
public:
    pmi_float& val (); // thermal conductivity
};

    PMI_MSC_ThermalConductivity_Base (const PMI_Environment& env,
                                      const std::string& msconfig_name,
                                      const int model_index,
                                      const std::string& model_string,
                                      const PMI_AnisotropyType
                                      anisotype);
    virtual ~PMI_MSC_ThermalConductivity_Base ();

    PMI_AnisotropyType AnisotropyType () const;

    virtual void compute (const Input& input, Output& output) = 0;
};
```

The prototype for the virtual constructor is given as:

```
typedef PMI_MSC_ThermalConductivity_Base*
new_PMI_MSC_ThermalConductivity_Base_func
(const PMI_Environment& env, const std::string& msconfig_name,
const int model_index, const std::string& model_string,
const PMI_AnisotropyType anisotype);
extern "C" new_PMI_MSC_ThermalConductivity_Base_func
new_PMI_MSC_ThermalConductivity_Base;
```

## Thermal Properties and Heat

This section presents the following PMIs:

- [Distributed Thermal Resistance](#)
  - [Heat Capacity](#)
  - [Heat Generation Rate](#)
  - [Metal Thermoelectric Power](#)
  - [Thermal Conductivity](#)
  - [Thermoelectric Power](#)
- 

### Distributed Thermal Resistance

For the distributed thermal resistance model, you can specify your own resistance model by using some physical variables at the two sides of the interface (see [Boundary Conditions for Lattice Temperature on page 283](#)). The syntax in the command file is:

```
Physics(MaterialInterface = "Silicon/Aluminum") {  
    DistrThermalResist = DTResistPMI(pmi_model_name)  
}
```

### Dependencies

The distributed thermal resistance model can depend on the following variables:

T1	Lattice temperature for material 1 [K]
T2	Lattice temperature for material 2 [K]
Tn1	Electron temperature for material 1 [K]
Tn2	Electron temperature for material 2 [K]
Tp1	Hole temperature for material 1 [K]
Tp2	Hole temperature for material 2 [K]
N1	Electron density for material 1 [ $\text{cm}^{-3}$ ]
N2	Electron density for material 2 [ $\text{cm}^{-3}$ ]

## Chapter 39: Physical Model Interface

### Thermal Properties and Heat

P1 Hole density for material 1 [cm<sup>-3</sup>]

P2 Hole density for material 2 [cm<sup>-3</sup>]

By default, you can control the material or region order by specifying `MaterialInterface` or `RegionInterface` in the `Physics` section. The material or region that is specified first uses the PMI variable labeled with 1 (`T1`, `N1`, `P1`, `Tn1`, and `Tp1`), and the second material or region of the interface uses the PMI variable set labeled with 2 (`T2`, `N2`, `P2`, `Tn2`, and `Tp2`). Taking the previous example, at the `Silicon/Aluminum` region interface, the PMI variable set 1 refers to the `Silicon` region, and set 2 refers to the `Aluminum` region. In addition, you can set the reference material or region with the keyword `reference`. For example:

```
Physics(MaterialInterface = "Silicon/Aluminum") {
    DistrThermalResist = DTResistPMI(
        pmi_model_name
        reference = Aluminum
    )
}
```

Here, the keyword `reference` forces the PMI variable set 1 for `Aluminum` regions and set 2 for `Silicon` regions, thereby overwriting the order specified with `MaterialInterface`.

## Simplified C++ Interface

The following base class is declared in the file `PMI.h`:

```
class PMI_DistrThermalResist_Base : public PMI_Vertex_Base {
public:
    class Input : public PMI_Vertex_Input_Base {
public:
    Input (const PMI_DistrThermalResist_Base* DistrThermResist_base,
           const int vertex);
    pmi_float t1; // lattice temperature for material1
    pmi_float tn1; // electron temperature for material1
    pmi_float tp1; // hole temperature for material1
    pmi_float n1; // electron density for material1
    pmi_float p1; // hole density for material1
    pmi_float t2; // lattice temperature for material2
    pmi_float tn2; // electron temperature for material2
    pmi_float tp2; // hole temperature for material2
    pmi_float n2; // electron density for material2
    pmi_float p2; // hole density for material2
};
    class Output {
public:
    pmi_float resist; // Schottky resistance
};
    PMI_DistrThermalResist_Base (const PMI_Environment& env);
    virtual ~PMI_DistrThermalResist_Base ();
}
```

## Chapter 39: Physical Model Interface

### Thermal Properties and Heat

```
    virtual void compute (const Input& input1, Output& output) = 0;  
};
```

The prototype for the virtual constructor is given as:

```
typedef PMI_DistrThermalResist_Base*  
    new_PMI_DistrThermalResist_Base_func  
    (const PMI_Environment& env);  
extern "C" PMI_DistrThermalResist_Base*  
    new_PMI_DistrThermalResist_Base  
    (const PMI_Environment& env) {  
        return new DTRPMI(env);  
}
```

## Example: Electron Density Dependence

A simple electron density-dependent model can be implemented as follows:

```
#include "PMI.h"  
  
class DTRPMI: public PMI_DistrThermalResist_Base {  
private:  
  
public:  
    DTRPMI(const PMI_Environment& env);  
    ~DTRPMI();  
    void compute(const Input& input1, Output& output);  
};  
  
// constructor  
DTRPMI::DTRPMI(const PMI_Environment&  
                 env):PMI_DistrThermalResist_Base(env) {  
}  
  
// destructor  
DTRPMI::~DTRPMI() {  
    // nothing to do here (nothing allocated)  
}  
  
void DTRPMI::compute(const Input& input1, Output& output) {  
    pmi_float T1 = input1.t1;  
    pmi_float Tn1 = input1.tn1;  
    pmi_float Tp1 = input1.tp1;  
    pmi_float N1 = input1.n1;  
    pmi_float P1 = input1.p1;  
    pmi_float T2 = input1.t2;  
    pmi_float Tn2 = input1.tn2;  
    pmi_float Tp2 = input1.tp2;  
    pmi_float N2 = input1.n2;  
    pmi_float P2 = input1.p2;  
    //output the distributed thermal resistance.  
    //Users can modify this part.  
    output.resist = 1e-17*(N1);
```

## Chapter 39: Physical Model Interface

### Thermal Properties and Heat

```
}
```

```
extern "C" PMI_DistrThermalResist_Base*
    new_PMI_DistrThermalResist_Base
(const PMI_Environment& env) {
    return new DTRPMI(env);
}
```

---

## Heat Capacity

The model for lattice heat capacity in [Equation 71 on page 248](#) can be specified in the `Physics` section of the command file. The following two possibilities are available:

```
Physics {
    HeatCapacity (
        constant
        pmi_model_name
    )
}
```

These entries have the following meaning:

<code>constant</code>	Use constant heat capacity (default)
<code>pmi_model_name</code>	Call a PMI model to compute the heat capacity

## Dependencies

The heat capacity  $c_L$  can depend on the variable:

$t$  Lattice temperature [K]

The PMI model must compute the following results:

$c$  Heat capacity  $c_L$  [ $\text{JK}^{-1}\text{cm}^{-3}$ ]

In the case of the standard interface, the following derivative must be computed as well:

$\frac{dc}{dt}$  Derivative of  $c_L$  with respect to  $t$  [ $\text{JK}^{-2}\text{cm}^{-3}$ ]

## Chapter 39: Physical Model Interface

### Thermal Properties and Heat

## Standard C++ Interface

The following base class is declared in the file `PMIModels.h`:

```
class PMI_HeatCapacity : public PMI_Vertex_Interface {  
  
public:  
    PMI_HeatCapacity (const PMI_Environment& env);  
  
    virtual ~PMI_HeatCapacity ();  
  
    virtual void Compute_c  
    (const double t, double& c) = 0;  
  
    virtual void Compute_dcdt  
    (const double t, double& dcdt) = 0  
};
```

The following virtual constructor must be implemented:

```
typedef PMI_HeatCapacity* new_PMI_HeatCapacity_func  
    (const PMI_Environment& env);  
extern "C" new_PMI_HeatCapacity_func new_PMI_HeatCapacity;
```

## Simplified C++ Interface

The following base class is declared in the file `PMI.h`:

```
class PMI_HeatCapacity_Base : public PMI_Vertex_Base {  
  
public:  
    class Input : public PMI_Vertex_Input_Base {  
    public:  
        pmi_float t; // lattice temperature  
    };  
  
    class Output {  
    public:  
        pmi_float c; // heat capacity  
    };  
  
    PMI_HeatCapacity_Base (const PMI_Environment& env);  
    virtual ~PMI_HeatCapacity_Base ();  
  
    virtual void compute (const Input& input, Output& output) = 0;  
};
```

The prototype for the virtual constructor is given as:

```
typedef PMI_HeatCapacity_Base* new_PMI_HeatCapacity_Base_func  
    (const PMI_Environment& env);  
extern "C" new_PMI_HeatCapacity_Base_func new_PMI_HeatCapacity_Base;
```

## Chapter 39: Physical Model Interface

### Thermal Properties and Heat

## Example: Constant Heat Capacity

The following C++ code implements constant heat capacity:

```
#include "PMIModels.h"

class Constant_HeatCapacity : public PMI_HeatCapacity {
private:
    double cv;

public:
    Constant_HeatCapacity (const PMI_Environment& env);
    ~Constant_HeatCapacity ();

    void Compute_c
        (const double t, double& c);

    void Compute_dcdt
        (const double t, double& dcdt);
};

Constant_HeatCapacity::
Constant_HeatCapacity (const PMI_Environment& env) :
    PMI_HeatCapacity (env)
{ // default values
    cv = InitParameter ("cv", 1.63);
}

Constant_HeatCapacity::
~Constant_HeatCapacity ()
{
}

void Constant_HeatCapacity::
Compute_c (const double t, double& c)
{ c = cv;
}

void Constant_HeatCapacity::
Compute_dcdt (const double t, double& dcdt)
{ dcdt = 0.0;
}

extern "C"
PMI_HeatCapacity* new_PMI_HeatCapacity
    (const PMI_Environment& env)
{ return new Constant_HeatCapacity (env);
}
```

## Heat Generation Rate

The total heat generation rate is the term on the right-hand side of [Equation 72 on page 249](#). You can specify an additional term `pmi_Heat` for a given material or region:

```
Total_Heat = existing_Heat + pmi_Heat
```

The name of the PMI must be specified in the new subsection `HeatSource(pmi_model)` in the `Physics` section of the command file as follows:

```
Physics (Material = "Silicon") {  
    HeatSource (pmi_Heat_Si)  
}
```

To plot the `pmi_Heat` values, specify the keyword `pmiHeat` in the `Plot` section of the command file.

## Dependencies

The heat generation depends on the variables:

- n        Electron density [ $\text{cm}^{-3}$ ]
- p        Hole density [ $\text{cm}^{-3}$ ]
- t        Lattice temperature [K]
- f        Electric field vector [ $\text{Vcm}^{-1}$ ]
- g        Gradient temperature [ $\text{Kcm}^{-1}$ ]

The PMI model must compute the following results:

`Heat`      Heat generation rate [ $\text{Wcm}^{-3}$ ]

In the case of the standard interface, the following derivatives must be computed as well:

`dHeat_dn`    Derivative of `Heat` with respect to `n` [W]

`dHeat_dp`    Derivative of `Heat` with respect to `p` [W]

`dHeat_dt`    Derivative of `Heat` with respect to `t` [ $\text{Wcm}^{-3}\text{K}^{-1}$ ]

## Chapter 39: Physical Model Interface

### Thermal Properties and Heat

dHeat\_df Derivative of Heat with respect to each component f [ $\text{Wcm}^{-2}\text{V}^{-1}$ ]

dHeat\_dg Derivative of Heat with respect to each component g [ $\text{Wcm}^{-2}\text{K}^{-1}$ ]

## Standard C++ Interface

The following base class is declared in the file PMIModels.h:

```
class PMI_Heat_Generation : public PMI_Vertex_Interface {

public:
    PMI_Heat_Generation (const PMI_Environment& env);
    virtual ~PMI_Heat_Generation () ;

    // methods to be implemented by user
    virtual void Compute_Heat
        (const double n,                      // electron density
         const double p,                      // hole density
         const double t,                      // lattice temperature
         const double f[3],                   // electric field vector
         const double g[3],                   // gradient of temperature
         double& heat) = 0;                  // heat generation rate

    virtual void Compute_dHeat_dn
        (const double n,                      // electron density
         const double p,                      // hole density
         const double t,                      // lattice temperature
         const double f[3],                   // electric field vector
         const double g[3],                   // gradient of temperature
         double& dHeat_dn) = 0;              // derivative of heat with respect to n

    virtual void Compute_dHeat_dp
        (const double n,                      // electron density
         const double p,                      // hole density
         const double t,                      // lattice temperature
         const double f[3],                   // electric field vector
         const double g[3],                   // gradient of temperature
         double& dHeat_dp) = 0;              // derivative of heat with respect to p

    virtual void Compute_dHeat_dt
        (const double n,                      // electron density
         const double p,                      // hole density
         const double t,                      // lattice temperature
         const double f[3],                   // electric field vector
         const double g[3],                   // gradient of temperature
         double& dHeat_dt) = 0;              // derivative of heat with respect to t

    virtual void Compute_dHeat_df
        (const double n,                      // electron density
         const double p,                      // hole density
```

## Chapter 39: Physical Model Interface

### Thermal Properties and Heat

```
const double t,           // lattice temperature
const double f[3],        // electric field vector
const double g[3],        // gradient of temperature
double dHeat_df[3]) = 0; // derivative of heat with respect to f

virtual void Compute_dHeat_dg
(const double n,          // electron density
 const double p,          // hole density
 const double t,          // lattice temperature
 const double f[3],        // electric field vector
 const double g[3],        // gradient of temperature
 double dHeat_dg[3]) = 0; // derivative of heat with respect to g
};
```

The following virtual constructor must be implemented:

```
typedef PMI_Heat_Generation* new_PMI_Heat_Generation_func
(const PMI_Environment& env);
extern "C" new_PMI_Heat_Generation_func new_PMI_HeatGeneration;
```

## Simplified C++ Interface

The following base class is declared in the file `PMI.h`:

```
class PMI_HeatGeneration_Base : public PMI_Vertex_Base {

public:
    class Input : public PMI_Vertex_Input_Base {
public:
    pmi_float n;      // electron density
    pmi_float p;      // hole density
    pmi_float t;      // lattice temperature
    pmi_float f[3];   // electric field vector
    pmi_float g[3];   // gradient of temperature
};

    class Output {
public:
    pmi_float heat; // heat generation rate
};

    PMI_HeatGeneration_Base (const PMI_Environment& env);
    virtual ~PMI_HeatGeneration_Base ();

    virtual void compute (const Input& input, Output& output) = 0;
};

}
```

The prototype for the virtual constructor is given as:

```
typedef PMI_HeatGenerationFunction_Base*
new_PMI_HeatGenerationFunction_Base_func
(const PMI_Environment& env);
```

## Chapter 39: Physical Model Interface

### Thermal Properties and Heat

```
extern "C" new_PMI_HeatGenerationFunction_Base_func  
new_PMI_HeatGenerationFunction_Base;
```

## Example: Dependency on Electric Field and Gradient of Temperature

In the following example, heat generation has a linear dependency on the electric field and the gradient of temperature:

```
// Heat = kappa*(E,gradT)*1[1/V]  
// where  
// E - electric field vector,  
// gradT - gradient of temperature.  
  
//  
// The 1D equation  
// -div(kappa*grad(T(x))) = Heat, 0 <= x <= L  
// T(0) = T0,  
// T(L) = T1,  
// has the exact solution:  
// T(x) = (T1-T0)*(exp(-E*x)-1)/(exp(-E*L)-1) + T0  
  
class HeatGeneration : public PMI_HeatGeneration_Base  
{  
private:  
    double kappa;  
  
public:  
    HeatGeneration (const PMI_Environment& env);  
    ~HeatGeneration ();  
    void compute (const Input& input, Output& output);  
};  
  
HeatGeneration::HeatGeneration (const PMI_Environment& env) :  
    PMI_HeatGeneration_Base (env)  
{  
    kappa = InitParameter("kappa", 0.01); // [W/(K*cm)]  
}  
  
HeatGeneration::~HeatGeneration () {}  
  
void HeatGeneration::compute (const Input& input, Output& output)  
{  
    const pmi_float (&f)[3] = input.f;  
    const pmi_float (&g)[3] = input.g;  
  
    output.heat = kappa*(f[0]*g[0] + f[1]*g[1] + f[2]*g[2]);  
}  
  
extern "C"  
PMI_HeatGeneration_Base* new_PMI_HeatGeneration_Base  
(const PMI_Environment& env)  
{ return new HeatGeneration (env); }
```

## Metal Thermoelectric Power

The metal thermoelectric power  $P$  in metals can be defined by a PMI (see [Thermoelectric Power on page 1032](#)).

The name of the PMI can be specified regionwise, materialwise, or globally in the `Physics` section of the command file as follows:

```
Physics(Material="Copper") {
    MetalTEPower(pmi_model)
}
```

## Dependencies

The metal thermoelectric power can depend on the following variables:

- $t$  Lattice temperature [K]
- $q_f$  Quasi-Fermi potential [V]
- $\mathbf{f}$  Electric field vector [ $V\text{cm}^{-1}$ ]

The PMI model must compute the following result:

- $\text{power}$  Thermoelectric power [ $\text{VK}^{-1}$ ]

In the case of the standard interface, the following derivatives must be computed as well:

- $d\text{power}/dt$  Derivative of  $\text{power}$  with respect to  $t$  [ $\text{VK}^{-2}$ ]
- $d\text{power}/dq_f$  Derivative of  $\text{power}$  with respect to  $q_f$  [ $\text{K}^{-1}$ ]
- $d\text{power}/d\mathbf{f}$  Derivative of  $\text{power}$  with respect to each component of  $\mathbf{f}$  [ $\text{cm K}^{-1}$ ], a vector with up to three entries

## Standard C++ Interface

The following base class is declared in the file `PMIModels.h`:

```
class PMI_MetalThermoElectricPower : public PMI_Vertex_Interface {

public:
    PMI_MetalThermoElectricPower(const PMI_Environment& env);
    virtual ~MetalPMI_ThermoElectricPower();
```

## Chapter 39: Physical Model Interface

### Thermal Properties and Heat

```
virtual void Compute_power(
    const double t,           // lattice temperature
    const double qf,          // quasi-Fermi potential
    const double f[3]          // electric field vector
    double& power);          // thermoelectric power

virtual void Compute_dpowerdt(
    const double t,           // lattice temperature
    const double qf,          // quasi-Fermi potential
    const double f[3]          // electric field vector
    double& dpowerdt);        // derivative of thermoelectric power
                            // with respect to lattice temperature

virtual void Compute_dpowerdqf(
    const double t,           // lattice temperature
    const double qf,          // quasi-Fermi potential
    const double f[3]          // electric field vector
    double& dpowerdqf);       // derivative of thermoelectric power
                            // with respect to quasi-Fermi potential

virtual void Compute_dpowerdf(
    const double t,           // lattice temperature
    const double qf,          // quasi-Fermi potential
    const double f[3]          // electric field vector
    double& dpowerdf[3]);     // derivative of thermoelectric power
                            // with respect to electric field
};
```

The following virtual constructor must be implemented:

```
typedef PMI_MetalThermoElectricPower*
    new_PMI_MetalThermoElectricPower_func
    (const PMI_Environment& env);
extern "C" new_PMI_MetalThermoElectricPower_func
    new_PMI_MetalThermoElectricPower;
```

## Simplified C++ Interface

The following base class is declared in the file `PMI.h`:

```
class PMI_MetalThermoElectricPower_Base : public PMI_Vertex_Base {

public:
    class Input : public PMI_Vertex_Input_Base {
public:
    pmi_float t;           // lattice temperature
    pmi_float qf;          // quasi-Fermi potential
    pmi_float f[3];         // electric field vector
};

    class Output {
public:
    pmi_float power; // thermoelectric power
};
```

## Chapter 39: Physical Model Interface

### Thermal Properties and Heat

```
};

PMI_MetalThermoElectricPower_Base (const PMI_Environment& env);
virtual ~PMI_MetalThermoElectricPower_Base ();

    virtual void compute (const Input& input, Output& output) = 0;
};
```

The following virtual constructor must be implemented:

```
extern "C"
PMI_MetalThermoElectricPower_Base*
    new_PMI_MetalThermoElectricPower_Base
(const PMI_Environment& env);
```

## Example: Linear Field Dependency of Metal TEP

The following C++ code implements a metal TEP PMI depending linearly on two of the electric field components:

```
#include "PMIModels.h"

namespace {
    // Boltzmann constant in J/K
    double kB = 1.380662e-23;
    // electron charge in C
    double e0 = 1.602192e-19;
}

class MetalThermoElectricPower : public PMI_MetalThermoElectricPower
{
private:
    double alpha, beta;

public:
    MetalThermoElectricPower(const PMI_Environment& env);
    ~MetalThermoElectricPower ();

    void Compute_power
        (const double t,           // lattice temperature
         const double qf,          // quasi-Fermi potential
         const double f[3],         // electric field vector
         double& power);          // thermoelectric power

    void Compute_dpowerdt
        (const double t,           // lattice temperature
         const double qf,          // quasi-Fermi potential
         const double f[3],         // electric field vector
         double& dpowerdt);        // derivative of thermoelectric power
                                    // with respect to lattice temperature

    void Compute_dpowerdqf
        (const double t,           // lattice temperature
```

## Chapter 39: Physical Model Interface

### Thermal Properties and Heat

```
        const double qf,           // quasi-Fermi potential
        const double f[3],         // electric field vector
        double& dpowerdqf);      // derivative of thermoelectric power
                                // with respect to quasi-Fermi
                                // potential

    void Compute_dpowerdf
        (const double t,          // lattice temperature
         const double qf,          // quasi-Fermi potential
         const double f[3],         // electric field vector
         double dpowerdf[3]);      // derivative of thermoelectric power
                                // with respect to electric field
    }

MetalThermoElectricPower::
MetalThermoElectricPower (const PMI_Environment& env) :
    PMI_MetalThermoElectricPower (env)
{
    // default values
    alpha = InitParameter("alpha",1);
    beta = InitParameter("beta",1e-2);
}

MetalThermoElectricPower::
~MetalThermoElectricPower ()
{
}

void MetalThermoElectricPower::
Compute_power (const double t, const double qf, const double f[3],
               double& power)
{
    power = 1e-6 * 1e-2 * (f[0] + f[1]);
}

void MetalThermoElectricPower::
Compute_dpowerdt (const double t, const double qf, const double f[3],
                  double& dpowerdt)
{
    dpowerdt = 0;
}

void MetalThermoElectricPower::
Compute_dpowerdqf (const double t, const double qf, const double f[3],
                   double& dpowerdqf)
{
    dpowerdqf = 0;
}

void MetalThermoElectricPower::
Compute_dpowerdf (const double t, const double qf, const double f[3],
                  double dpowerdf[3])
{
```

## Chapter 39: Physical Model Interface

### Thermal Properties and Heat

```
dpowerdf[0] = 1e-6 * 1e-2;
dpowerdf[1] = 1e-6 * 1e-2;
dpowerdf[2] = 0;
}

extern "C"
PMI_MetalThermoElectricPower* new_PMI_MetalThermoElectricPower
  (const PMI_Environment& env)
{
  return new MetalThermoElectricPower(env);
}
```

---

## Thermal Conductivity

The PMI provides access to the lattice thermal conductivity  $\kappa$  in [Equation 71 on page 248](#). To activate it, in the `Physics` section of the command file, specify:

```
Physics {
  ThermalConductivity ( <string> )
}
```

where the string is the name of the PMI model.

The PMI supports anisotropic thermal conductivity, and the model can be evaluated along different crystallographic axes. The enumeration type `PMI_AnisotropyType` as defined in [Mobility on page 1278](#) determines the axis. The default is isotropic thermal conductivity. If anisotropic thermal conductivity is activated in the command file, then the PMI class `PMI_ThermalConductivity` is also instantiated in the anisotropic direction.

## Dependencies

The thermal conductivity  $\kappa$  can depend on the variable:

$t$  Lattice temperature [K]

The PMI model must compute the following results:

$\kappa$  Thermal conductivity  $\kappa$  [ $\text{Wcm}^{-1}\text{K}^{-1}$ ]

In the case of the standard interface, the following derivative must be computed as well:

$d\kappa/dt$  Derivative of  $\kappa$  with respect to  $t$  [ $\text{Wcm}^{-1}\text{K}^{-2}$ ]

## Chapter 39: Physical Model Interface

### Thermal Properties and Heat

## Standard C++ Interface

The following base class is declared in the file `PMIModels.h`:

```
class PMI_ThermalConductivity : public PMI_Vertex_Interface {

public:
    PMI_ThermalConductivity (const PMI_Environment& env,
                           const PMI_AnisotropyType anisotype);

    virtual ~PMI_ThermalConductivity () ;

    PMI_AnisotropyType AnisotropyType () const { return anisoType; }

    virtual void Compute_kappa
        (const double t, double& kappa) = 0;

    virtual void Compute_dkappadt
        (const double t, double& dkappadt) = 0;
};
```

The following virtual constructor must be implemented:

```
typedef PMI_ThermalConductivity* new_PMI_ThermalConductivity_func
    (const PMI_Environment& env, const PMI_AnisotropyType anisotype);
extern "C" new_PMI_ThermalConductivity_func
    new_PMI_ThermalConductivity;
```

## Simplified C++ Interface

The following base class is declared in the file `PMI.h`:

```
class PMI_ThermalConductivity_Base : public PMI_Vertex_Base {

public:
    class Input : public PMI_Vertex_Input_Base {
    public:
        pmi_float t;      // lattice temperature
    };

    class Output {
    public:
        pmi_float kappa; // thermal conductivity
    };

    PMI_ThermalConductivity_Base (const PMI_Environment& env,
                                 const PMI_AnisotropyType anisotype);

    virtual ~PMI_ThermalConductivity_Base ();

    PMI_AnisotropyType AnisotropyType () const;
```

## Chapter 39: Physical Model Interface

### Thermal Properties and Heat

```
    virtual void compute (const Input& input, Output& output) = 0;  
};
```

The prototype for the virtual constructor is given as:

```
typedef PMI_ThermalConductivity_Base*  
        new_PMI_ThermalConductivity_Base_func  
    (const PMI_Environment& env, const PMI_AnisotropyType anisotype);  
extern "C" new_PMI_ThermalConductivity_Base_func  
new_PMI_ThermalConductivity_Base;
```

## Example: Temperature-Dependent Thermal Conductivity

The following C++ code implements the temperature-dependent thermal conductivity:

$$\kappa(T) = \frac{1}{a + bT + cT^2} \quad (1346)$$

as given in [Equation 1094 on page 1019](#).

```
#include "PMIModels.h"  
  
class TempDep_ThermalConductivity : public PMI_ThermalConductivity {  
private:  
    double a, b, c;  
  
public:  
    TempDep_ThermalConductivity (const PMI_Environment& env, const  
        PMI_AnisotropyType anisotype);  
    ~TempDep_ThermalConductivity ();  
  
    void Compute_kappa  
        (const double t, double& kappa);  
  
    void Compute_dkappadt  
        (const double t, double& dkappadt);  
};  
  
TempDep_ThermalConductivity::  
TempDep_ThermalConductivity (const PMI_Environment& env, const  
    PMI_AnisotropyType anisotype) :  
    PMI_ThermalConductivity (env, anisotype)  
{ // default values  
    a = InitParameter ("a", 0.03);  
    b = InitParameter ("b", 1.56e-03);  
    c = InitParameter ("c", 1.65e-06);  
}  
  
TempDep_ThermalConductivity::  
~TempDep_ThermalConductivity ()  
{
```

## Chapter 39: Physical Model Interface

### Thermal Properties and Heat

```
}
```

```
void TempDep_ThermalConductivity::  
Compute_kappa (const double t, double& kappa)  
{ kappa = 1.0 / (a + b*t + c*t*t);  
}
```

```
void TempDep_ThermalConductivity::  
Compute_dkappadt (const double t, double& dkappadt)  
{ const double kappa = 1.0 / (a + b*t + c*t*t);  
    dkappadt = -kappa * kappa * (b + 2.0*c*t);  
}
```

```
extern "C"  
PMI_ThermalConductivity* new_PMI_ThermalConductivity  
    (const PMI_Environment& env, const PMI_AnisotropyType anisotype)  
{ return new TempDep_ThermalConductivity (env, anisotype);  
}
```

## Example: Thin-Layer Thermal Conductivity

In this example, thermal conductivity depends on the doping and thickness of layers. In this case, the external `LayerThickness` command (see [Extracting Layer Thickness on page 379](#)) must be specified in the command file:

```
Physics (Material="Silicon") {  
    LayerThickness(<parameters>)    # external command for thickness  
    extraction  
    ThermalConductivity (ThinLayerKappa)  
}
```

The following C++ code implements the thin-layer doping-dependent thermal conductivity (see `$STROOT/tcad/$STRELEASE/lib/sdevice/src/pmi_ThinLayerKappa/ThinLayerKappa.C`):

$$\kappa = \kappa_0 \cdot \kappa_d(d) \cdot \kappa_h(h) \quad (1347)$$

where:

- $\kappa_0$  is a constant value [ $\text{Wcm}^{-1}\text{K}^{-1}$ ].
- $d$  is Doping/ScaleDoping (unitless doping).
- $h$  is LayerThickness/ScaleThickness (unitless layer thickness).
- $\kappa_d(d) = (a_2(d-d_0)^2 + a_1(d-d_0) + a_0) \cdot \exp(\alpha(d-d_0))$ .
- $\kappa_h(h) = (b_2(h-h_0)^2 + b_1(h-h_0) + b_0) \cdot \exp(\beta(h-h_0))$ .

## Chapter 39: Physical Model Interface

### Thermal Properties and Heat

```
#include <math.h>
#include "PMI.h"

class ThinLayer_ThermalConductivity : public
    PMI_ThermalConductivity_Base
{
private:
    double kappa0;                      // [W/(K*cm)]
    double ScaleDoping;                 // [cm-3]
    double ScaleThickness;               // [um]
    double d0, a0, a1, a2, alpha;       // [1]
    double h0, b0, b1, b2, beta;        // [1]

public:
    ThinLayer_ThermalConductivity (const PMI_Environment& env,
                                    const PMI_AnisotropyType anisotype);
    ~ThinLayer_ThermalConductivity () ;

    void compute (const Input& input, Output& output);
};

ThinLayer_ThermalConductivity:::
ThinLayer_ThermalConductivity (const PMI_Environment& env,
                               const PMI_AnisotropyType anisotype) :
    PMI_ThermalConductivity_Base (env, anisotype)
{ // default values
    kappa0 = InitParameter("kappa0", 1.);           // [W/(K*cm)]

    ScaleDoping = InitParameter("ScaleDoping", 1.e+18);   // [cm-3]
    d0      = InitParameter("d0", 1.);                // [1]
    a0      = InitParameter("a0", 1.);                // [1]
    a1      = InitParameter("a1", 0.);                // [1]
    a2      = InitParameter("a2", 0.);                // [1]
    alpha   = InitParameter("alpha", 0.);              // [1]

    ScaleThickness = InitParameter("ScaleThickness", 1.e-3); // [um] =
                                                               // 1 nm
    h0      = InitParameter("h0", 1.);                // [1]
    b0      = InitParameter("b0", 1.);                // [1]
    b1      = InitParameter("b1", 0.);                // [1]
    b2      = InitParameter("b2", 0.);                // [1]
    beta   = InitParameter("beta", 0.);              // [1]
}

ThinLayer_ThermalConductivity:::
~ThinLayer_ThermalConductivity ()
{ }

void ThinLayer_ThermalConductivity:::
compute (const Input& input, Output& output)
{
    const double h  = input.ReadLayerThickness()/ScaleThickness;
    pmi_float Nd = input.ReadDoping(PMI_Donor);
```

## Chapter 39: Physical Model Interface

### Thermal Properties and Heat

```
pmi_float Na = input.ReadDoping(PMI_Acceptor);
pmi_float d = (Nd - Na)/ScaleDoping;

pmi_float kd = (a2*(d-d0)*(d-d0) + a1*(d-d0) +
                 a0)*exp(alpha*(d-d0));
pmi_float kh = (b2*(h-h0)*(h-h0) + b1*(h-h0) +
                 b0)*exp(beta*(h-h0));

pmi_float kappa = kappa0*kd*kh;

output.kappa = kappa; // [W/(K*cm)]
}

extern "C"
PMI_ThermalConductivity_Base* new_PMI_ThermalConductivity_Base
    (const PMI_Environment& env, const PMI_AnisotropyType anisotype)
{ return new ThinLayer_ThermalConductivity (env, anisotype);
}
```

The following example is a parameter file for the formula  $\kappa = (2d^2 + d + 1) \cdot \exp(0.1 \cdot h)$ :

```
Material = "Silicon" {

    ThinLayerKappa {
        kappa0 = 1 # [W/(K*cm)]

        ScaleDoping = 1.e+18 # [cm^-3]
        d0      = 0      # [1]
        a0      = 1      # [1]
        a1      = 1      # [1]
        a2      = 2      # [1]
        alpha   = 0      # [1]

        ScaleThickness = 1.e-3 # [um] = 1 nm
        h0      = 0      # [1]
        b0      = 1      # [1]
        b1      = 0      # [1]
        b2      = 0      # [1]
        beta   = 0.1    # [1]
    }
}
```

---

## Thermoelectric Power

The thermoelectric powers  $P_n$  and  $P_p$  in semiconductors can be defined by a PMI (see [Thermoelectric Power on page 1032](#)).

The name of the PMI can be specified regionwise, materialwise, or globally in the `Physics` section of the command file as follows:

```
Physics(Material="Silicon") { TEPower(pmi_model) }
```

## Chapter 39: Physical Model Interface

### Thermal Properties and Heat

## Dependencies

The semiconductor TEPs can depend on the following variables:

$t$  Lattice temperature [K]

$\text{dens}$  Carrier density [ $\text{cm}^{-3}$ ]

The PMI model must compute the following results:

$\text{power}$  Thermoelectric power [ $\text{VK}^{-1}$ ]

In the case of the standard interface, the following derivatives must be computed as well:

$\text{dpowerdt}$  Derivative of  $\text{power}$  with respect to  $t$  [ $\text{VK}^{-2}$ ]

$\text{dpowerddens}$  Derivative of  $\text{power}$  with respect to  $\text{dens}$  [ $\text{VK}^{-1}\text{cm}^{-3}$ ]

## Standard C++ Interface

The following base class is declared in the file `PMIModels.h`:

```
class PMI_ThermoElectricPower : public PMI_Vertex_Interface {  
  
public:  
    PMI_ThermoElectricPower(const PMI_Environment& env);  
    virtual ~PMI_ThermoElectricPower();  
  
    virtual void Compute_power(  
        const double t,           // lattice temperature  
        const double dens,        // carrier density  
        double& power);         // thermoelectric power  
  
    virtual void Compute_dpowerdt(  
        const double t,           // lattice temperature  
        const double dens,        // carrier density  
        double& dpowerdt);       // derivative of thermoelectric power  
                           // with respect to lattice temperature  
  
    virtual void Compute_dpowerddens(  
        const double t,           // lattice temperature  
        const double dens,        // carrier density  
        double& dpowerddens);    // derivative of thermoelectric power  
                           // with respect to carrier density  
};
```

## Chapter 39: Physical Model Interface

### Thermal Properties and Heat

The following virtual constructor must be implemented:

```
typedef PMI_ThermoElectricPower* new_PMI_ThermoElectricPower_func
    (const PMI_Environment& env);
extern "C" new_PMI_ThermoElectricPower_func
    new_PMI_e_ThermoElectricPower;
extern "C" new_PMI_ThermoElectricPower_func
    new_PMI_h_ThermoElectricPower;
```

## Simplified C++ Interface

The following base class is declared in the file `PMI.h`:

```
class PMI_ThermoElectricPower_Base : public PMI_Vertex_Base {

public:
    class Input : public PMI_Vertex_Input_Base {
    public:
        pmi_float t;           // lattice temperature
        pmi_float dens;        // carrier density
    };

    class Output {
    public:
        pmi_float power; // thermoelectric power
    };

    PMI_ThermoElectricPower_Base (const PMI_Environment& env);
    virtual ~PMI_ThermoElectricPower_Base () ;

    virtual void compute (const Input& input, Output& output) = 0;
};
```

The following virtual constructor must be implemented:

```
extern "C"
PMI_ThermoElectricPower_Base* new_PMI_ThermoElectricPower_Base
    (const PMI_Environment& env);
```

## Example: Analytic TEP

The following C++ code implements an analytic TEP model (see [Thermoelectric Power on page 1032](#)):

```
#include "PMIModels.h"

namespace {
    // pi
    double pi = 3.141592654;
    // electron mass in kg
    double m0 = 9.109534e-31;
    // Planck's constant in J*s
    double h_planck = 6.626176e-34;
```

## Chapter 39: Physical Model Interface

### Thermal Properties and Heat

```
// Boltzmann constant in J/K
double kB = 1.380662e-23;
// electron charge in C
double e0 = 1.602192e-19;

double pow_1_5 (double x) {
    return (x==1) ? 1 : x * sqrt (x);
}

double Compute_NB_By3_2 (double m_r) {
    double val = 2 * pow_1_5(2 * pi * (kB/h_planck) *
                           (m0/h_planck)) / 1e6;
    val *= pow_1_5(m_r);
    return val;
}

class AnalyticalTEP_ThermoElectricPower : public
    PMI_ThermoElectricPower
{
protected:
    double k_c, s_c;
    // relative effective mass
    double m_r;
    int sign;

public:
    AnalyticalTEP_ThermoElectricPower(const PMI_Environment& env);
    ~AnalyticalTEP_ThermoElectricPower ();

    void Compute_power
        (const double t,          // lattice temperature
         const double dens,       // carrier density
         double& power);        // thermoelectric power

    void Compute_dpowerdt
        (const double t,          // lattice temperature
         const double dens,       // carrier density
         double& dpowerdt);     // derivative of thermoelectric power
                                // with respect to lattice temperature

    void Compute_dpoderddens
        (const double t,          // lattice temperature
         const double dens,       // carrier density
         double& dpoderddens);   // derivative of thermoelectric power
                                // with respect to carrier density
};

AnalyticalTEP_ThermoElectricPower::
AnalyticalTEP_ThermoElectricPower (const PMI_Environment& env) :
    PMI_ThermoElectricPower (env)
{}
```

## Chapter 39: Physical Model Interface

### Thermal Properties and Heat

```
AnalyticalTEP_ThermoElectricPower::  
~AnalyticalTEP_ThermoElectricPower ()  
{  
}  
  
void AnalyticalTEP_ThermoElectricPower::  
Compute_power (const double t, const double dens, double& power)  
{  
    double NB = Compute_NB_By3_2(m_r) * pow_1_5(t);  
    if(sign < 0)  
        power = k_c * (log(dens/NB) + s_c - 2.5);  
    else  
        power = k_c * (log(NB/dens) - s_c + 2.5);  
    power *= (kB/e0);  
}  
  
void AnalyticalTEP_ThermoElectricPower::  
Compute_dpowerdt (const double t, const double dens, double& dpowerdt)  
{  
    dpowerdt = sign * (kB/e0) * k_c * 1.5 / t;  
}  
  
void AnalyticalTEP_ThermoElectricPower::  
Compute_dpowerddens (const double t, const double dens, double&  
                     dpowerddens)  
{  
    dpowerddens = -sign * (kB/e0) * k_c / dens;  
}  
  
class AnalyticalTEP_e_ThermoElectricPower :  
public AnalyticalTEP_ThermoElectricPower  
{  
public:  
    AnalyticalTEP_e_ThermoElectricPower(const PMI_Environment& env);  
    ~AnalyticalTEP_e_ThermoElectricPower () {}  
};  
  
AnalyticalTEP_e_ThermoElectricPower::  
AnalyticalTEP_e_ThermoElectricPower(const PMI_Environment& env) :  
AnalyticalTEP_ThermoElectricPower(env) {  
    // default values  
    k_c = InitParameter("k_c_e", 1);  
    s_c = InitParameter("s_c_e", 1);  
    m_r = InitParameter("m_r_e", 1);  
    sign = -1;  
}  
  
class AnalyticalTEP_h_ThermoElectricPower :  
public AnalyticalTEP_ThermoElectricPower  
{  
public:  
    AnalyticalTEP_h_ThermoElectricPower(const PMI_Environment& env);
```

## Chapter 39: Physical Model Interface

### Optics

```
~AnalyticalTEP_h_ThermoElectricPower () {}  
};  
  
AnalyticalTEP_h_ThermoElectricPower::  
AnalyticalTEP_h_ThermoElectricPower(const PMI_Environment& env) :  
AnalyticalTEP_ThermoElectricPower(env) {  
    // default values  
    k_c = InitParameter("k_c_h", 1);  
    s_c = InitParameter("s_c_h", 1);  
    m_r = InitParameter("m_r_h", 1);  
    sign = 1;  
}  
  
extern "C"  
PMI_ThermoElectricPower* new_PMI_e_ThermoElectricPower  
(const PMI_Environment& env)  
{  
    return new AnalyticalTEP_e_ThermoElectricPower(env);  
}  
  
extern "C"  
PMI_ThermoElectricPower* new_PMI_h_ThermoElectricPower  
(const PMI_Environment& env)  
{  
    return new AnalyticalTEP_h_ThermoElectricPower(env);  
}
```

---

## Optics

This section presents the following PMIs:

- [Complex Refractive Index Model Interface](#)
- [Optical Quantum Yield](#)
- [Special Contact PMI for Raytracing](#)

---

## Complex Refractive Index Model Interface

The complex refractive index model interface (CRIMI) allows the addition of new complex refractive index models as a function of almost any internally available variable. These models must be implemented as C++ functions, and Sentaurus Device loads the functions at runtime using the dynamic loader. No access to the Sentaurus Device source code is necessary. The concept is similar to that of the PMI; however, it is not limited to Sentaurus Device.

The generated shared object containing the model implementation can be used together with Sentaurus Device Electromagnetic Wave Solver (see *Sentaurus™ Device*

## Chapter 39: Physical Model Interface

### Optics

*Electromagnetic Wave Solver User Guide*, Complex Refractive Index Model Interface) and Sentaurus Mesh (see *Sentaurus™ Mesh User Guide*, Computing Cell Size Automatically (EMW Applications)).

Three main steps are required for integrating user-defined models:

1. A C++ class implementing the complex refractive index model must be written.
2. A shared object must be created that can be loaded at runtime.
3. The model must be activated in the command file.

These steps are described in the following sections in more detail.

## C++ Application Programming Interface

For each complex refractive index model, two C++ subroutines must be implemented: one to compute the refractive index and one to compute the extinction coefficient. More specifically, a C++ class must be implemented that is derived from a base class declared in the header file `CRIModels.h`. In addition, you must provide a virtual constructor function, which allocates an instance of the derived class, and a virtual destructor function, which deallocates it.

The public interface of the base class from which a model-specific class must be derived is declared in the header file `CRIModels.h` as:

```
class CRI_Model : public CRI_Model_Interface {  
public:  
    CRI_Model(const CRI_Environment& env);  
    virtual ~CRI_Model();  
    //definition of complex refractive index  
    struct ComplexRefractiveIndexConstituentsReal {  
        double n0;  
        double dn_lambda;  
        double dn_temp;  
        double dn_carr;  
        double dn_gain;  
    };  
  
    struct ComplexRefractiveIndexConstituentsImag {  
        double k0;  
        double dk_lambda;  
        double dk_carr;  
    };  
  
    //methods to be implemented by user  
  
    //definition of complex refractive index in terms of its constituents  
    virtual void Compute_n(ComplexRefractiveIndexConstituentsReal& data);  
    virtual void Compute_k(ComplexRefractiveIndexConstituentsImag& data);
```

## Chapter 39: Physical Model Interface

### Optics

```
//direct definition of complex refractive index
virtual void Compute_n(double& n) = 0;
virtual void Compute_k(double& k) = 0;
};
```

The CRIMI provides a *basic interface* where only the actual value of the refractive index and the extinction coefficient can be overridden as well as an *advanced interface*. In the advanced interface, the total values are computed automatically from its constituents, and it has the advantage that the user-specified constituents can be visualized instead of its default values. If carrier dependency is modeled with a CRIMI and quantum yield or temperature or both should be consistent, the advanced interface must be used. If a CRIMI is used in a thermal simulation where free carrier absorption plays a role, using the advanced interface is mandatory. For details, see [Quantum Yield Models on page 658](#).

The arguments of the `Compute` functions indicate that they are passed as references to the respective functions. They carry the values corresponding to the model specification in the `ComplexRefractiveIndex` section in the command file. When implementing a user-defined complex refractive index model, the values for `n` and `k` (in the case of the basic interface) or the members of `struct ComplexRefractiveIndexConstituentsReal` and `struct ComplexRefractiveIndexConstituentsImag` must be overwritten in the function body. Otherwise, the original values remain unchanged, which can be useful if, for example, only a new model for either the real or the imaginary part of the complex refractive index must be implemented or if only a specific constituent needs to be modified.

Often, a complex refractive index model is based on several material-specific parameters. For each region or material, these parameters can be defined in special sections of the parameter file that carry the model name as given in the command file. The parameters are best initialized in the constructor as it is called once for every region.

For this purpose, the member function called `InitParameter` is used, which has two arguments. The first argument is the name of the parameter as listed in the parameter file, and the second argument is its default value. The latter is used if no value is assigned to the parameter for a particular region or material. The supported value types for user-defined parameters are integer, floating-point number, and string.

The following example shows how to initialize a parameter called `Gamma` in the constructor:

```
Constant_CRI_Model::Constant_CRI_Model(const CRI_Environment& env) :
CRI_Model(env) {

    Gamma = InitParameter("Gamma", 1.5);

}
```

where `Gamma` was declared as a data member of type `double` in the class `Constant_CRI_Model`.

## Chapter 39: Physical Model Interface

### Optics

In addition to the implementation of the two class member functions `Compute_n` and `Compute_k`, the virtual constructor function and the virtual destructor function must be defined as:

```
extern "C" {

    opto_n_cri::CRI_Model*
    new_CRI_Model(const opto_n_cri::CRI_Environment& env)
    {
        return new opto_n_cri::Constant_CRI_Model(env);
    }

    void
    delete_CRI_Model(opto_n_cri::CRI_Model* cri_model)
    {
        delete cri_model;
    }

}
```

In these sample functions, it is assumed that the name of the user-provided derived class is `Constant_CRI_Model`.

### Runtime Support

The base class of `CRI_Model` is derived from another class called `CRI_Model_Interface`, which adds several functions that extend the possibilities for defining new complex refractive index models. Among them are functions that query basic properties of the model such as its name, for which region and material it is called and which models have been specified in the command file.

Another group of functions allows you to read the values of the most common variables on which the complex refractive index depends, namely, wavelength, carrier densities, and temperature as well as the default values for the various contributions to the complex refractive index. For most models, this set of functions should be sufficient.

For advanced models, it might be necessary to have access to additional internal variables to model the dependencies correctly. To this end, three interface functions are available that allow you to query, to register, and to read the internal variables available. A specific variable can only be read in the `Compute` functions if it has been registered in the constructor. The internal name of the corresponding variable can be obtained from the function `GetAvailableVariables`, which returns a vector of strings containing the names of all supported variables.

The remaining functions offer support for mole fraction-dependent models and for direct access of the `NumericalTables` defined in the parameter file.

The signatures of the runtime support functions discussed here can be found in the header file `CRIModels.h` and are briefly described below. Corresponding type declarations are

## Chapter 39: Physical Model Interface

### Optics

contained in the header file `ExportedTypes.h` in the same installation directory. These header files are located in `$STROOT/tcad/$STRELEASE/lib/opto/include/`.

General utility functions:

- `std::string Name() const`: Returns the name of `CRIModel` as specified in the command file.
- `std::string ReadRegionName() const`: Returns the name of the region to which the vertex belongs.
- `std::string ReadMaterialName() const`: Returns the name of the material to which the vertex belongs.
- `bool WithModel(ComplexRefractiveIndexModelType model) const`: Returns `true` if `model` is activated for the region to which the vertex belongs; otherwise, `false`. For each `model`, a corresponding specification in the `ComplexRefractiveIndex` section of the command file exists (see [Using Complex Refractive Index on page 697](#)).

Functions for reading the values of the most common variables on which the complex refractive index depends:

- `double ReadWavelength() const`: Returns wavelength in [ $\mu\text{m}$ ]
- `double ReadTemperature() const`: Returns temperature in [K]
- `double Read_eDensity() const`: Returns electron density in [ $\text{cm}^{-3}$ ]
- `double Read_hDensity() const`: Returns hole density in [ $\text{cm}^{-3}$ ]
- `double ReadQW_eDensity() const`: Returns quantum-well electron density corresponding to the bound states of the well in [ $\text{cm}^{-3}$ ]
- `double ReadQW_hDensity() const`: Returns quantum-well hole density corresponding to the bound states of the well in [ $\text{cm}^{-3}$ ]

Functions for accessing additional internal variables to model the dependencies correctly:

- `const std::vector<std::string>& GetAvailableVariables() const`: Returns a vector of strings containing all internal variable names.
- `void RegisterVariableToRead(std::string variable_name)`: Use variable name string from previous call to `GetAvailableVariables()` to register a variable for reading in the function body of the `Compute` functions. This is performed typically in the constructor.
- `double ReadVariableValue(std::string variable_name) const`: Returns current value of variable selected by specifying variable name string from previous call to `GetAvailableVariables()` as argument.

## Chapter 39: Physical Model Interface

### Optics

The following C++ code sample shows the use of the functions for accessing additional internal variables:

```
Constant_CRI_Model::Constant_CRI_Model(const CRI_Environment& env) :  
CRI_Model(env) {  
    // print out available variables  
    const std::vector<std::string>& variables = GetAvailableVariables();  
    std::cout << "Available variables: "  
    for(size_t i=0;i<variables.size();++i){  
        std::cout << variables[i] << " "  
    }std::cout << std::endl;  
  
    RegisterVariableToRead( "my_dn" );  
}  
  
void Constant_CRI_Model::Compute_n(double& n) {  
    double my_dataset_val = ReadVariableValue("my_dn");  
    n = Read_n();  
    n += my_dataset_val;  
}
```

In this code sample, the dataset `my_dn` is registered in the constructor as a readable variable and, in the `Compute` function, its value is added to the default refractive index given by the specified models such as `WavelengthDep` in the `ComplexRefractiveIndex` section. The `for` loop in the constructor is optional and allows you to query the exact names of the datasets, which are needed as arguments to the `ReadVariableValue` function.

Functions for reading the real and imaginary parts of the complex refractive index as well as its corresponding constituents computed according to the command file and parameter file specification:

- `double Read_n() const`: Returns the real part of the complex refractive index
- `double Read_k() const`: Returns the imaginary part of the complex refractive index
- `double Read_n0() const`: Returns the base refractive index given in [Equation 704 on page 694](#)
- `double Read_k0() const`: Returns the base extinction coefficient given in [Equation 705 on page 694](#)
- `double Read_d_n_lambda() const`: Returns the change in refractive index due to its wavelength dependency
- `double Read_d_k_lambda() const`: Returns the change in extinction coefficient due to its wavelength dependency
- `double Read_d_n_temp() const`: Returns the change in refractive index due to its temperature dependency

## Chapter 39: Physical Model Interface

### Optics

- `double Read_d_n_carr() const`: Returns the change in refractive index due to its carrier dependency
- `double Read_d_k_carr() const`: Returns the change in extinction coefficient due to its carrier dependency
- `double Read_d_n_gain() const`: Returns the change in extinction coefficient due to its gain dependency

Functions for retrieving information about mole fraction-dependent models and for direct access of the NumericalTables defined in the parameter file:

- `int ReadNumberOfMoleFractionIntervals() const`: Returns the number of mole-fraction intervals defined in the ComplexRefractiveIndex section of the parameter file.
- `double ReadxMoleFractionUpperLimit(int interval) const`: Returns upper limit of x-mole fraction interval specified as an argument.
- `double ReadNumericalTableValue_n(int interval=0) const`: Returns the refractive index for the current wavelength and the mole-fraction interval specified as an argument.
- `double ReadNumericalTableValue_k(int interval=0) const`: Returns the extinction coefficient for the current wavelength and the mole-fraction interval specified as an argument.
- `NumericalTable<double>* ReadNumericalTable(int interval=0) const`: Returns the numeric table for the mole-fraction interval specified as an argument. This function can be useful if a custom table interpolation algorithm must be implemented.

## Shared Object Code

Sentaurus Device assumes that the shared object code corresponding to a CRI model can be found in the file `modelname.so.arch`. The base name of this file must be identical to the name of the CRI model. The extension `.arch` depends on the hardware architecture.

You can use the script `cmi`, which is also a part of the CMI (see *Compact Models User Guide*, Chapter 4), to produce shared object files (see *Compact Models User Guide*, Runtime Support).

## Command File of Sentaurus Device

To load CRI models into Sentaurus Device, the `PMIPath` search path must be defined in the `File` section of the command file. The value of `PMIPath` consists of a sequence of directories. For example:

```
File { PMIPath = ". /home/brown/lib /home/jones/sdevice/lib" }
```

## Chapter 39: Physical Model Interface

### Optics

For each CRI model, which appears in the `ComplexRefractiveIndex` section, the given directories are searched for a corresponding shared object file `modelname.so.arch`.

A CRI model can be activated in the `ComplexRefractiveIndex` section of the command file by specifying the name of the model as shown in the following example:

```
Physics {
    Optics (
        ComplexRefractiveIndex (
            CRIModel (Name = "modelname")
        )
    )
}
```

A CRI model name can only consist of alphanumeric characters and underscores (`_`). The first character must be either a letter or an underscore. All the CRI models can be specified regionwise or materialwise:

```
Physics (region = "Region.1") { ... }
Physics (material = "SiO2") { ... }
```

when using the unified interface for optical generation computation in Sentaurus Device; otherwise, only global specification is supported.

---

## Optical Quantum Yield

The quantum yield factors  $\eta_G$ ,  $\eta_{T_{Eg}}$ , and  $\eta_{T_0}$  defined by [Equation 691 on page 660](#) can be accessed through a PMI.

In the command file, the PMI is specified in the `QuantumYield` section as follows:

```
Physics {
    Optics (
        OpticalGeneration (
            QuantumYield (
                ...
                pmiModel = "<pmi_model_name>"
            )
        )
    )
}
```

## Dependencies

The quantum yield factors  $\eta_G$ ,  $\eta_{T_{Eg}}$ , and  $\eta_{T_0}$  can depend on the variables:

n                   Electron density [cm<sup>-3</sup>]

p                   Hole density [cm<sup>-3</sup>]

## Chapter 39: Physical Model Interface

### Optics

T	Lattice temperature [K]
bg	Bandgap energy $E_g$ [eV]
bg_eff	Effective bandgap energy (includes bandgap narrowing) $E_{g,eff}$ [eV]
wavelength	Wavelength of incident light $\lambda$ [ $\mu\text{m}$ ]
cplxRefIndex	Refractive index
cplxExtCoeff	Extinction coefficient
n_0	Base refractive index
k_0	Base extinction coefficient
d_n_lambda	Wavelength-dependent part of refractive index
d_k_lambda	Wavelength-dependent part of extinction coefficient
d_n_temp	Temperature-dependent part of refractive index
d_n_carr	Carrier-dependent part of refractive index
d_k_carr	Carrier-dependent part of extinction coefficient
d_n_gain	Gain-dependent part of refractive index

The PMI model must compute the following results:

eta_G	Quantum yield
eta_T_Eg	Thermalization yield (band gap)
eta_T0	Thermalization yield (vacuum)

## Standard C++ Interface

The following base class (public interface) is declared in the file `PMIModels.h`:

```
class PMI_OpticalQuantumYield : public PMI_Vertex_Interface
{
public:
    // the input data coming from the simulator
```

## Chapter 39: Physical Model Interface

### Optics

```
class idata {
public:
    idata(const void* );
    double n() const;
    double p() const;
    double T() const;
    double bg() const;
    double bg_eff() const;
    double wavelength() const;
    double cplxRefIndex() const;
    double cplxExtCoeff() const;
    double n_0() const;
    double k_0() const;
    double d_n_lambda() const;
    double d_k_lambda() const;
    double d_n_temp() const;
    double d_n_carr() const;
    double d_k_carr() const;
    double d_n_gain() const;
};

// the results computed by the PMI
class odata {
public:
    odata(void* );
    double& eta_G();
    double& eta_T_Eg();
    double& eta_T0();
};

// constructor and destructor
PMI_OpticalQuantumYield(const PMI_Environment& env);
virtual ~PMI_OpticalQuantumYield();

// compute value and derivatives
virtual void compute(const idata* id, odata* od ) = 0;
};
```

The prototype for the virtual constructor is given as:

```
typedef PMI_OpticalQuantumYield* new_PMI_OpticalQuantumYield_func
    (const PMI_Environment& env);
extern "C" new_PMI_OpticalQuantumYield_func
            new_PMI_OpticalQuantumYield;
```

## Simplified C++ Interface

The following base class (public interface) is declared in the file `PMI.h`:

```
class PMI_OpticalQuantumYield_Base : public PMI_Vertex_Base {
public:
    class Input : public PMI_Vertex_Input_Base {
public:
```

## Chapter 39: Physical Model Interface

### Optics

```
Input (const PMI_OpticalQuantumYield_Base*
       opticalquantumyield_base, const int vertex);
pmi_float n;
pmi_float p;
pmi_float T;
pmi_float bg;
pmi_float bg_eff;
pmi_float wavelength;
pmi_float cplxRefIndex;
pmi_float cplxExtCoeff;
pmi_float n_0;
pmi_float k_0;
pmi_float d_n_lambda;
pmi_float d_k_lambda;
pmi_float d_n_temp;
pmi_float d_n_carr;
pmi_float d_k_carr;
pmi_float d_n_gain;
};

class Output {
public:
    pmi_float eta_G;
    pmi_float eta_T_Eg;
    pmi_float eta_T0;
};

PMI_OpticalQuantumYield_Base (const PMI_Environment& env);
virtual ~PMI_OpticalQuantumYield_Base ();
virtual void compute (const Input& input, Output& output) = 0;
};
```

The prototype for the virtual constructor is given as:

```
typedef PMI_OpticalQuantumYield_Base*
new_PMI_OpticalQuantumYield_Base_func
(const PMI_Environment& env);
extern "C" new_PMI_OpticalQuantumYield_Base_func
new_PMI_OpticalQuantumYield_Base;
```

---

## Special Contact PMI for Raytracing

The PMI for raytracing allows you to access and change the parameters of a ray at special contacts. These contacts are drawn in the same manner as electrodes and thermodes (see [Boundary Condition for Raytracing on page 713](#)). The raytrace PMI is specified in the `RayTraceBC` section of the command file:

```
RayTraceBC {...  
    { Name = "pmi_contact" PMIModel = "pmi_modelname" }  
}
```

## Chapter 39: Physical Model Interface

### Optics

The name of "pmi\_contact" must match the contact name in the device. Any ray that hits this special contact invokes a call to this PMI. This raytrace PMI works only with the raytracer (see [Raytracer on page 702](#)) and the complex refractive index model (see [Complex Refractive Index Model on page 694](#)).

#### Note:

When using multithreading of the raytracer, you must carefully design the PMI code to be thread safe. This means that global variables should not be used since multiple threads might change the global variables at the same time, leading to confusion in the user PMI code at runtime.

## Dependencies

The raytrace PMI depends on the following variables:

wavelength	Wavelength [cm]
incident_angle	Incident angle [radian]
*incident_dirvec	Direction vector of incident ray
*polarvec	Polarization vector of incident ray
*normalvec	Normal vector to surface of impingement from region 1 to region 2
*intersectpoint	Intersection position vector [cm]
*region1_name	Name of region 1 (string)
*region2_name	Name of region 2 (string)
n1_real	Real part of refractive index 1
n1_imag	Imaginary part of refractive index 1
n2_real	Real part of refractive index 2
n2_imag	Imaginary part of refractive index 2
reflected_angle	Reflected angle [radian]
transmitted_angle	Transmitted angle [radian]
*reflected_dirvec	Direction vector of reflected ray
*transmitted_dirvec	Direction vector of transmitted ray

## Chapter 39: Physical Model Interface

### Optics

*reflected_startposition	Starting position vector of reflected ray [cm]
*transmitted_startposition	Starting position vector of transmitted ray [cm]
R_TE	Power TE reflection coefficient
T_TE	Power TE transmission coefficient
R_TM	Power TM reflection coefficient
T_TM	Power TM transmission coefficient

To obtain the rate intensity (units of  $s^{-1}$ ) carried by the ray, you need to compute the square of the length of \*polarvec. This rate intensity includes all previous absorptions sustained by the ray when it traversed absorptive regions of the device,<sup>1</sup> and it can be used directly to compute the amount of optical generation (with units of  $s^{-1}$ ). However, for raytracing used in LED simulations, the square of the length of \*polarvec gives only a relative intensity value because the polarization vector of the head ray of the raytree is initialized to a length of 1.0.

If the reflection or transmission coefficients are equal to zero, then no reflected or transmitted rays are created in the raytracing process.

With this PMI model, you can change the following variables:

reflected_angle	Reflected angle [radian]
transmitted_angle	Transmitted angle [radian]
*reflected_dirvec	Direction vector of reflected ray
*transmitted_dirvec	Direction vector of transmitted ray
*reflected_startposition	Starting position vector of reflected ray [cm]
*transmitted_startposition	Starting position vector of transmitted ray [cm]
R_TE	Power TE reflection coefficient
T_TE	Power TE transmission coefficient
R_TM	Power TM reflection coefficient
T_TM	Power TM transmission coefficient

If you want to change the direction vector, angle, or position vector of the reflected or transmitted ray, then the respective flags must be set to TRUE:

- is\_reflectedangle\_changed
- is\_reflecteddirvec\_changed
- is\_transmittedangle\_changed
- is\_transmitteddirvec\_changed
- is\_reflected\_new\_startposition
- is\_transmitted\_new\_startposition

However, no flags are needed if you want to change the power reflection or transmission coefficients.

## Standard C++ Interface

The following base class is declared in the file `PMIModels.h`:

```
class PMI_RayTraceBoundary : public PMI_Vertex_Interface {
public:
    PMI_RayTraceBoundary (const PMI_Environment& env);
    virtual ~PMI_RayTraceBoundary();

    // methods to be implemented by user
    virtual void Compute_BoundaryParameters
        ( // Non-changeable quantities
        const double wavelength,           // wavelength [cm]
        const double incident_angle,      // incident angle
        const double* incident_dirvec,   // dir vector of incident ray
        const double* polarvec,          // polar vec of incident ray
        const double* normalvec,         // normal to impingement
        const double* intersectpoint,    // intersection point
        const char* region1_name,        // name of region 1
        const char* region2_name,        // name of region 2
        const double n1_real,            // real part of refr index 1
        const double n1_imag,            // imag part of refr index 1
        const double n2_real,            // real part of refr index 2
        const double n2_imag,            // imag part of refr index 2
        // User changeable quantities
        bool& is_reflectedangle_changed, // is refl angle changed?
        bool& is_reflecteddirvec_changed, // is reflected dir changed?
        bool& is_transmittedangle_changed, // is transm angle changed?
        bool& is_transmitteddirvec_changed, // is transm dir changed?
        bool& is_reflected_new_startposition, // is refl pos changed?
        bool& is_transmitted_new_startposition, // is transm pos
                                              // changed?
        double& reflected_angle,        // reflected angle
        double& transmitted_angle,      // transmitted angle
```

## Chapter 39: Physical Model Interface

### Optics

```
    double* reflected_dirvec,           // dir vec of reflected
          // ray
    double* transmitted_dirvec,         // dir vec of
          // transmitted ray
    double* reflected_startposition,   // start pos of
          // reflected ray
    double* transmitted_startposition, // start pos of transm
          // ray
    double& R_TE,                   // power TE reflection coeff.
    double& T_TE,                   // power TE transmission coeff.
    double& R_TM,                   // power TM reflection coeff.
    double& T_TM                     // power TM transmission coeff.
) = 0;

// Auxiliary functions for users
void ReadComplexRefractiveIndex(std::string location_name,
    double wavelength,      // in microns
    double& n,
    double& k,
    PMI_RayTraceBoundary::LocationType location_type =
    PMI_RayTraceBoundary::Material);
private:
    const PMI_Environment* thisenv;
};
```

An internal auxiliary function called `ReadComplexRefractiveIndex(...)` allows you to compute the complex refractive index of any material or region at a particular wavelength. The constraint is that, if a requested material does not exist in the device structure, then it must at least be defined in the parameter file. The complex refractive index specification (for example, wavelength dependency for the real part or imaginary part or both) is taken from the command file. For materials that do not exist in the device structure, the specification from the default `Physics` section is used unless it is explicitly given in a corresponding separate material `Physics` section.

## Example: Assessing and Modifying a Ray

The following example shows how to access the information about a ray that intersects the special raytrace PMI contact, and how you can change the information of the ray:

```
class Dummy_RayTraceBoundary : public PMI_RayTraceBoundary {
private:
    // short ind_field; // field index for optical generation
    //     double shape; // transient curve shape
    //     double G1, G2, G3, T1, T2, T3, T4; // const for shapes
    int count;
    double d1;
public:
    Dummy_RayTraceBoundary (const PMI_Environment& env);
    ~Dummy_RayTraceBoundary ();

    void Compute_BoundaryParameters
```

## Chapter 39: Physical Model Interface

### Optics

```
(const double wavelength,           // wavelength [cm]
 const double incident_angle,      // incident angle
 const double* incident_dirvec,    // dir vec of incident ray
 const double* polarvec,          // polar vec of incident ray
 const double* normalvec,         // normal to impingement
 const double* intersectpoint,     // intersection point
 const char* region1_name,        // name of region 1
 const char* region2_name,        // name of region 2
 const double n1_real,            // real part of refr index 1
 const double n1_imag,            // imag part of refr index 1
 const double n2_real,            // real part of refr index 2
 const double n2_imag,            // imag part of refr index 2
 // User changeable quantities
 bool& is_reflectedangle_changed, // is refl angle changed?
 bool& is_reflecteddirvec_changed, // is refl dir changed?
 bool& is_transmittedangle_changed, // is transm angle changed?
 bool& is_transmitteddirvec_changed, // is transm dir changed?
 bool& is_reflected_new_startposition, // is refl pos changed?
 bool& is_transmitted_new_startposition, // is transm pos
                                         // changed?
 double& reflected_angle,        // reflected angle
 double& transmitted_angle,       // transmitted angle
 double* reflected_dirvec,       // dir vec of reflected ray
 double* transmitted_dirvec,     // dir vec of transmitted ray
 double* reflected_startposition, // start pos of reflected ray
 double* transmitted_startposition, // start pos of transm ray
 double& R_TE,                  // power TE reflection coeff.
 double& T_TE,                  // power TE transmission
                                         // coeff.
 double& R_TM,                  // power TM reflection coeff.
 double& T_TM,                  // power TM transmission
                                         // coeff.
 );
};

Dummy_RayTraceBoundary::  
Dummy_RayTraceBoundary (const PMI_Environment& env) :  
    PMI_RayTraceBoundary (env)  
{  
    printf("PMI: initializing ray trace PMI\n");  
}  
  
Dummy_RayTraceBoundary::  
~Dummy_RayTraceBoundary ()  
{  
}  
  
void Dummy_RayTraceBoundary::  
Compute_BoundaryParameters (  
    const double wavelength,  
    const double incident_angle,  
    const double* incident_dirvec,  
    const double* polarvec,
```

## Chapter 39: Physical Model Interface

### Optics

```
const double* normalvec,
const double* intersectpoint,
const char* region1_name,
const char* region2_name,
const double n1_real,
const double n1_imag,
const double n2_real,
const double n2_imag,
// User changeable quantities
bool& is_reflectedangle_changed,
bool& is_reflecteddirvec_changed,
bool& is_transmittedangle_changed,
bool& is_transmitteddirvec_changed,
bool& is_reflected_new_startposition,
bool& is_transmitted_new_startposition,
double& reflected_angle,
double& transmitted_angle,
double* reflected_dirvec,
double* transmitted_dirvec,
double* reflected_startposition,
double* transmitted_startposition,
double& R_TE,
double& T_TE,
double& R_TM,
double& T_TM
)
{
    // Ray goes from region 1 to region 2.
    // PMI contact is the interface between regions 1 and 2.
    printf("Region 1: Name=%s, Refractive Index = %e + i%e\n",
        region1_name, n1_real, n1_imag);
    printf("Angles: Incident=%lf, Reflected=%lf, Transmitted=%lf\n",
        incident_angle, reflected_angle, transmitted_angle);
    printf("Wavelength = %e [cm]\n", wavelength);
    printf("Incident Ray Direction=(%e,%e,%e)\n",
        incident_dirvec[0], incident_dirvec[1], incident_dirvec[2]);
    printf("Power Coefficients: R_TE=%e, T_TE=%e, R_TM=%e, T_TM=%e\n",
        R_TE, T_TE, R_TM, T_TM);

    // For example, change reflected direction to (1,2,3)
    is_reflecteddirvec_changed = TRUE;
    reflected_dirvec[0] = 1.0;
    reflected_dirvec[1] = 2.0;
    reflected_dirvec[2] = 3.0;
}
extern "C" {
    PMI_RayTraceBoundary* new_PMI_RayTraceBoundary (const
        PMI_Environment& env)
    { return new Dummy_RayTraceBoundary (env);
    }
}
```

## Mechanical Stress

This section presents the following PMIs:

- [Mobility Stress Factor](#)
  - [Piezoelectric Polarization](#)
  - [Piezoresistive Coefficients](#)
  - [Stress](#)
- 

### Mobility Stress Factor

Stress-dependent isotropic mobility enhancement factors can be computed by a mobility stress factor PMI. These factors are applied to total low-field mobility or mobility components as described in [Isotropic Factor Models on page 988](#).

The name of a mobility stress factor PMI model is specified as a `Factor` option in the command file (see [Using Isotropic Factor Models on page 989](#)):

```
Physics {
    Piezo (
        Model (
            Mobility (
                Factor (
                    pmi_model_name
                    [ChannelDirection=<n>]
                    [AutoOrientation | ParameterSetName="<psname>"]
                )
            )
        )
    )
}
```

If specified, then the `Factor` options `ChannelDirection=<n>`, `AutoOrientation`, and `ParameterSetName="<psname>"` are passed as parameters to the mobility stress factor PMI model (`AutoOrientation` is passed as `AutoOrientation=1`).

### Dependencies

A mobility stress factor PMI model can depend on the following variable:

`enorm`      Normal electric field [ $\text{Vcm}^{-1}$ ]

## Chapter 39: Physical Model Interface

### Mechanical Stress

The PMI model must compute the following result:

`mobilitystressfactor` Stress-dependent mobility enhancement factor [1]

In the case of the standard interface, the following derivative must be computed as well:

`dmobilitystressfactordenorm` Derivative of the mobility stress factor with respect to  $E_{\text{normal}}$  [ $\text{cm V}^{-1}$ ]

## Standard C++ Interface

The following base class is declared in the file `PMIModels.h`:

```
class PMI_MobilityStressFactor : public PMI_Vertex_Interface {  
  
public:  
    PMI_MobilityStressFactor (const PMI_Environment& env);  
  
    virtual ~PMI_MobilityStressFactor ();  
  
    virtual void Compute_mobilitystressfactor  
        (const double enorm, double& mobilitystressfactor,  
         double& dmobilitystressfactordenorm) = 0;  
};
```

Two virtual constructors are required for electron and hole mobility factors:

```
typedef PMI_MobilityStressFactor* new_PMI_MobilityStressFactor_func  
    (const PMI_Environment& env);  
extern "C" new_PMI_MobilityStressFactor_func  
    new_PMI_e_MobilityStressFactor;  
extern "C" new_PMI_MobilityStressFactor_func  
    new_PMI_h_MobilityStressFactor;
```

## Simplified C++ Interface

The following base class is declared in the file `PMI.h`:

```
class PMI_MobilityStressFactor_Base : public PMI_Vertex_Base {  
  
public:  
    class Input : public PMI_Vertex_Input_Base {  
    public:  
        pmi_float enorm; // normal to interface electric field  
    };  
  
    class Output {  
    public:
```

## Chapter 39: Physical Model Interface

### Mechanical Stress

```
    pmi_float mobilitystressfactor; // mobility enhancement stress
                                    // factor
};

PMI_MobilityStressFactor_Base (const PMI_Environment& env);
virtual ~PMI_MobilityStressFactor_Base ();
virtual void compute (const Input& input, Output& output) = 0;
};
```

The prototype for the virtual constructor is given as:

```
typedef PMI_MobilityStressFactor_Base*
new_PMI_MobilityStressFactor_Base_func
(const PMI_Environment& env);
extern "C" new_PMI_MobilityStressFactor_Base_func
new_PMI_e_MobilityStressFactor_Base;
extern "C" new_PMI_MobilityStressFactor_Base_func
new_PMI_h_MobilityStressFactor_Base;
```

## Example: Effective Stress

This example illustrates the implementation of the `EffectiveStressModel` (see [Effective Stress Model on page 990](#)) using the standard C++ interface:

```
#include "PMI.h"
#include <cmath>

class pmi_EffectiveStressModel : public PMI_MobilityStressFactor {

protected:
    int cd, dim;
    double alpha1, alpha2, alpha3, beta11, beta12,
           beta13, beta22, beta23, beta33;
    double mu0, a10, a11, a12, a20, a21, a22, s00, s01, s02, t0, t1, t2, F0;

public:
    pmi_EffectiveStressModel (const PMI_Environment& env);
    ~pmi_EffectiveStressModel ();
    void Compute_mobilitystressfactor (const double enorm,
                                       double& mobilitystressfactor,
                                       double& dmobilitystressfactordenorm);
};

pmi_EffectiveStressModel::
pmi_EffectiveStressModel (const PMI_Environment& env) :
    PMI_MobilityStressFactor (env)
{
}

pmi_EffectiveStressModel::
~pmi_EffectiveStressModel ()
{
}
void pmi_EffectiveStressModel::
```

## Chapter 39: Physical Model Interface

### Mechanical Stress

```
Compute_mobilitystressfactor (const double enorm,
                               double& mobilitystressfactor,
                               double& dmobilitystressfactordenorm)
{
    // Get stress components and convert to MPa.
    double stress[6];
    stress[0] = 1.e-6*ReadStress(PMI_StressXX);
    stress[1] = 1.e-6*ReadStress(PMI_StressYY);
    stress[2] = 1.e-6*ReadStress(PMI_StressZZ);
    stress[3] = 1.e-6*ReadStress(PMI_StressYZ);
    stress[4] = 1.e-6*ReadStress(PMI_StressXZ);
    stress[5] = 1.e-6*ReadStress(PMI_StressXY);

    // Get the diagonal stresses used in the calculation.
    double S11 = 0.0, S22 = 0.0, S33 = 0.0;

    // 1D cases and 2D cases where cd != 2.
    if (dim == 1 || (dim == 2 && cd <2)) {
        const int nd = (cd == 0) ? 1 : 0;
        const int pd = 3-cd-nd;
        S11 = stress[cd];
        S22 = stress[nd];
        S33 = stress[pd];
    }

    // Otherwise, do a stress transformation to get stress components.
    else {
        // Normal direction vector.
        double norm[3];
        ReadNearestInterfaceNormal(norm[0], norm[1], norm[2]);
        norm[cd] = 0.0;
        if (norm[0] == 0. && norm[1] == 0. && norm[2] == 0.0) {
            if (cd == 0) norm[1] = 1.0;
            else norm[0] = 1.0;
        }
        else {
            const double mag=sqrt(norm[0]*norm[0]+norm[1]*norm[1]
                                  +norm[2]*norm[2]);
            norm[0] /= mag; norm[1] /= mag; norm[2] /= mag;
        }

        // Channel direction vector.
        double chan[3] = {0.0};
        chan[cd] = 1.0;

        // In-plane direction vector.
        double plan[3];
        plan[0] = chan[1]*norm[2] - chan[2]*norm[1];
        plan[1] = chan[2]*norm[0] - chan[0]*norm[2];
        plan[2] = chan[0]*norm[1] - chan[1]*norm[0];

        // Rotation matrix.
        double a[3][3];
```

## Chapter 39: Physical Model Interface

### Mechanical Stress

```
a[0][0]=chan[0]; a[0][1]=chan[1]; a[0][2]=chan[2];
a[1][0]=norm[0]; a[1][1]=norm[1]; a[1][2]=norm[2];
a[2][0]=plan[0]; a[2][1]=plan[1]; a[2][2]=plan[2];

// Get the diagonal components of the transformed stress tensor.
double SD[3][3];
SD[0][0] = stress[0]; SD[1][1] = stress[1]; SD[2][2] = stress[2];
SD[0][1] = SD[1][0] = stress[5];
SD[0][2] = SD[2][0] = stress[4];
SD[1][2] = SD[2][1] = stress[3];

// Channel direction.
S11 = SD[cd][cd];

// Normal direction.
for (int i=0; i<3; ++i) {
    for (int j=0; j<3; ++j) {
        S22 += a[1][i]*a[1][j]*SD[i][j];
    }
}

// In-plane direction.
for (int i=0; i<3; ++i) {
    for (int j=0; j<3; ++j) {
        S33 += a[2][i]*a[2][j]*SD[i][j];
    }
}

// Effective stress.
const double Seff = alpha1*S11 + alpha2*S22 + alpha3*S33
+ beta11*S11*S11 + beta12*S11*S22 + beta13*S11*S33
+ beta22*S22*S22 + beta23*S22*S33 + beta33*S33*S33;

// Convert field to MV/cm.
const double F = 1.e-6*enorm;

// Get the stress factor.
const double FF = (F < F0) ? F : F0;
const double F2 = FF*FF;
const double A1 = a10 + a11*FF + a12*F2;
const double A2 = a20 + a21*FF + a22*F2;
const double S0 = s00 + s01*FF + s02*F2;
const double t = t0 + t1*F + t2*F*F;
const double sarg = (Seff - S0)/t;
const double expsarg = exp(sarg);
const double expsargp1 = 1. + expsarg;
mobilitystressfactor = ((A1-A2)/expsargp1 + A2)/mu0;

// Derivative wrt enorm.
const double dA1dF = (F < F0) ? a11 + 2.*a12*F : 0.0;
const double dA2dF = (F < F0) ? a21 + 2.*a22*F : 0.0;
const double dS0dF = (F < F0) ? s01 + 2.*s02*F : 0.0;
```

## Chapter 39: Physical Model Interface

### Mechanical Stress

```
const double dtdF = t1 + 2.*t2*F;
const double dsfdF = (dA1dF + expsarg*dA2dF + (A1-A2)*expsarg*
                      (dS0dF + sarg*dtdF)/(t*expsargp1))/(mu0*expsargp1);
dmobilitystressfactordenorm = 1.e-6*dsfdF;
}

class pmi_e_EffectiveStressModel : public pmi_EffectiveStressModel {
public:
    pmi_e_EffectiveStressModel (const PMI_Environment& env);
    ~pmi_e_EffectiveStressModel () {}
};

pmi_e_EffectiveStressModel:::
pmi_e_EffectiveStressModel (const PMI_Environment& env)
: pmi_EffectiveStressModel (env)
{
    // Get channel direction and dimension.
    const double dchannelDirection = InitParameter("ChannelDirection",
        1.0);
    cd = (dchannelDirection == 2.0) ? 1 : ((dchannelDirection ==
        3.0) ? 2 : 0);
    dim = ReadDimension();

    alpha1 = InitParameter ("alpha1_e", 1.0);
    alpha2 = InitParameter ("alpha2_e", -1.7);
    alpha3 = InitParameter ("alpha3_e", 0.7);
    beta11 = InitParameter ("beta11_e", 0.0);
    beta12 = InitParameter ("beta12_e", 0.0);
    beta13 = InitParameter ("beta13_e", 0.0);
    beta22 = InitParameter ("beta22_e", 0.0);
    beta23 = InitParameter ("beta23_e", 0.0);
    beta33 = InitParameter ("beta33_e", 0.0);
    mu0 = InitParameter ("mu0_e", 810.0);
    a10 = InitParameter ("a10_e", 565.0);
    a11 = InitParameter ("a11_e", -81.0);
    a12 = InitParameter ("a12_e", -44.0);
    a20 = InitParameter ("a20_e", 2028.0);
    a21 = InitParameter ("a21_e", -1992.0);
    a22 = InitParameter ("a22_e", 920.0);
    s00 = InitParameter ("s00_e", 1334.0);
    s01 = InitParameter ("s01_e", -2646.0);
    s02 = InitParameter ("s02_e", 875.0);
    t0 = InitParameter ("t0_e", 882.0);
    t1 = InitParameter ("t1_e", -987.0);
    t2 = InitParameter ("t2_e", 604.0);
    F0 = InitParameter ("F0_e", 1.e10);
}

class pmi_h_EffectiveStressModel : public pmi_EffectiveStressModel {
public:
    pmi_h_EffectiveStressModel (const PMI_Environment& env);
    ~pmi_h_EffectiveStressModel () {}
};
```

## Chapter 39: Physical Model Interface

### Mechanical Stress

```
pmi_h_EffectiveStressModel::  
pmi_h_EffectiveStressModel (const PMI_Environment& env)  
    : pmi_EffectiveStressModel (env)  
{  
    // Get channel direction and dimension.  
    const double dchannelDirection = InitParameter ("ChannelDirection",  
        1.0);  
    cd = (dchannelDirection == 2.0) ? 1 : ((dchannelDirection == 3.0) ?  
        2 : 0);  
    dim = ReadDimension();  
  
    alpha1 = InitParameter ("alpha1_h", 1.0);  
    alpha2 = InitParameter ("alpha2_h", -0.4);  
    alpha3 = InitParameter ("alpha3_h", -0.6);  
    beta11 = InitParameter ("beta11_h", 0.0);  
    beta12 = InitParameter ("beta12_h", 0.0);  
    beta13 = InitParameter ("beta13_h", -0.00004);  
    beta22 = InitParameter ("beta22_h", 0.00006);  
    beta23 = InitParameter ("beta23_h", -0.00018);  
    beta33 = InitParameter ("beta33_h", 0.00011);  
    mu0 = InitParameter ("mu0_h", 212.0);  
    a10 = InitParameter ("a10_h", 2460.0);  
    a11 = InitParameter ("a11_h", 0.0);  
    a12 = InitParameter ("a12_h", 0.0);  
    a20 = InitParameter ("a20_h", 42.0);  
    a21 = InitParameter ("a21_h", 0.0);  
    a22 = InitParameter ("a22_h", 0.0);  
    s00 = InitParameter ("s00_h", -1338.0);  
    s01 = InitParameter ("s01_h", 0.0);  
    s02 = InitParameter ("s02_h", 0.0);  
    t0 = InitParameter ("t0_h", 524.0);  
    t1 = InitParameter ("t1_h", 0.0);  
    t2 = InitParameter ("t2_h", 0.0);  
    F0 = InitParameter ("F0_h", 1.e10);  
}  
  
extern "C"  
PMI_MobilityStressFactor* new_PMI_e_MobilityStressFactor  
    (const PMI_Environment& env)  
{ return new pmi_e_EffectiveStressModel (env);  
}  
  
extern "C"  
PMI_MobilityStressFactor* new_PMI_h_MobilityStressFactor  
    (const PMI_Environment& env)  
{ return new pmi_h_EffectiveStressModel (env);
```

## Piezoelectric Polarization

The effects of piezoelectric polarization can be modeled by adding the divergence of the piezoelectric polarization vector as an additional charge term:

$$q_{\text{PE}} = -\nabla \cdot P_{\text{PE}} \quad (1348)$$

to the right-hand side of the Poisson equation (see [Equation 37 on page 229](#)):

$$\nabla \cdot \epsilon \cdot \nabla \phi = -q(p - n + N_D - N_A) - q_{\text{PE}} \quad (1349)$$

The quantity  $P_{\text{PE}}$  denotes the piezoelectric polarization vector, which can be defined by a PMI. The built-in models for piezoelectric polarization are discussed in [Dependency of Saturation Velocity on Stress on page 997](#).

The name of the PMI is specified in the `Physics` section of the command file as follows:

```
Physics {
    Piezoelectric_Polarization (pmi_polarization)
}
```

The piezoelectric polarization vector and the piezoelectric charge can be plotted by:

```
Plot {
    PE_Polarization/vector
    PE_Charge
}
```

Sentaurus Device assumes that the piezoelectric polarization vector  $P_{\text{PE}}$  is zero outside of the device. This boundary condition might lead to an unexpectedly large charge density if  $P_{\text{PE}}$  has a nonzero component orthogonal to the boundary (discontinuity in  $\nabla P_{\text{PE}}$ ).

## Dependencies

The piezoelectric polarization model does not have explicit dependencies. However, it can use the runtime support. In particular, it has access to the stress fields.

The model must compute:

`pol` Piezoelectric polarization vector [ $\text{C cm}^{-2}$ ]

The resulting vector `pol` has the dimension 3. However, only the first `dim` components need to be defined, where `dim` is equal to the dimension of the problem.

## Chapter 39: Physical Model Interface

### Mechanical Stress

## Standard C++ Interface

The following base class is declared in the file `PMIModels.h`:

```
class PMI_Polarization : public PMI_Vertex_Interface {  
  
public:  
    PMI_Polarization (const PMI_Environment& env);  
    virtual ~PMI_Polarization ();  
  
    virtual void Compute_pol  
        (double pol [3]) = 0;  
};
```

The prototype for the virtual constructor is given as:

```
typedef PMI_Polarization* new_PMI_Polarization_func  
    (const PMI_Environment& env);  
extern "C" new_PMI_Polarization_func new_PMI_Polarization;
```

## Simplified C++ Interface

The following base class is declared in the file `PMI.h`:

```
class PMI_Polarization_Base : public PMI_Vertex_Base {  
  
public:  
    class Input : public PMI_Vertex_Input_Base {  
    public:  
    };  
  
    class Output {  
    public:  
        pmi_float pol [3]; // piezoelectric polarization  
    };  
  
    PMI_Polarization_Base (const PMI_Environment& env);  
    virtual ~PMI_Polarization_Base ();  
  
    virtual void compute (const Input& input, Output& output) = 0;  
};
```

The prototype for the virtual constructor is given as:

```
typedef PMI_Polarization_Base* new_PMI_Polarization_Base_func  
    (const PMI_Environment& env);  
extern "C" new_PMI_Polarization_Base_func new_PMI_Polarization_Base;
```

## Chapter 39: Physical Model Interface

### Mechanical Stress

## Example: Gaussian Polarization

In this example, the piezoelectric polarization vector  $P_{PE}$  has a simple Gaussian shape in the x-direction:

```
#include "PMIModels.h"

class Gauss_Polarization : public PMI_Polarization {
private:
    double x0, c, a;

public:
    Gauss_Polarization (const PMI_Environment& env);
    ~Gauss_Polarization ();
    void Compute_pol (double pol [3]);
};

Gauss_Polarization::Gauss_Polarization (const PMI_Environment& env) :
    PMI_Polarization (env)
{ x0 = InitParameter ("x0", 0.0);
  c = InitParameter ("c", 1.0);
  a = InitParameter ("a", 1e-5);
}

Gauss_Polarization::~Gauss_Polarization ()
{
}

void Gauss_Polarization::Compute_pol (double pol [3])
{ double x, y, z;
  ReadCoordinate (x, y, z);
  pol [0] = a * exp (-c * (x-x0) * (x-x0));
  pol [1] = 0.0;
  pol [2] = 0.0;
}

extern "C"
PMI_Polarization* new_PMI_Polarization
    (const PMI_Environment& env)
{ return new Gauss_Polarization (env);
}
```

---

## Piezoresistive Coefficients

Sentaurus Device provides a PMI for implementing the dependencies of the piezoresistive prefactors over the normal electric field (see [Enormal- and MoleFraction-Dependent Piezo Coefficients on page 981](#)).

## Chapter 39: Physical Model Interface

### Mechanical Stress

It is activated in the `Piezo` section of the command file, in the `Tensor` subsection:

```
Physics {
    ...
    Piezo(
        Model(Mobility(Tensor("pmi_model")))
    )
}
```

## Dependencies

The piezoresistive prefactors  $e_{Pij}$  and  $h_{Pij}$  can depend on the following variable:

$E_{normal}$  Normal to interface semiconductor–dielectric electric field [ $V\text{cm}^{-1}$ ]

The PMI model must compute the following results:

$p_{11}, p_{12}, p_{44}$  Piezoresistive prefactors [1]

In the case of the standard interface, the following derivatives must be computed as well:

$dp_{11}, dp_{12}, dp_{44}$  Derivatives of  $p_{ij}$  with respect to  $E_{normal}$  [ $V^{-1}\text{cm}$ ]

## Standard C++ Interface

The following base class is declared in the file `PMIModels.h`:

```
class PMI_PiezoresistanceFactor : public PMI_Vertex_Interface {

public:
    PMI_PiezoresistanceFactor (const PMI_Environment& env);
    virtual ~PMI_PiezoresistanceFactor () ;

    virtual void Compute_Pij(const double Enormal,
                           double& p11, double& p12, double& p44) = 0;

    virtual void Compute_DerPij(const double Enormal,
                               double& dp11, double& dp12, double& dp44) = 0;
};
```

Two virtual constructors are required for the calculation of the piezoresistive prefactors.

```
typedef PMI_PiezoresistanceFactor* new_PMI_PiezoresistanceFactor_func
    (const PMI_Environment& env);
extern "C" new_PMI_PiezoresistanceFactor_func
    new_PMI_ePiezoresistanceFactor;
extern "C" new_PMI_PiezoresistanceFactor_func
    new_PMI_hPiezoresistanceFactor;
```

## Simplified C++ Interface

The following base class is declared in the file `PMI.h`:

```
class PMI_PiezoresistanceFactor_Base : public PMI_Vertex_Base {  
  
public:  
    class Input : public PMI_Vertex_Input_Base {  
public:  
    pmi_float Enormal; // normal to interface electric field  
};  
  
    class Output {  
public:  
    pmi_float p11; // piezoresistive prefactor  
    pmi_float p12; // piezoresistive prefactor  
    pmi_float p44; // piezoresistive prefactor  
};  
  
    PMI_PiezoresistanceFactor_Base (const PMI_Environment& env);  
    virtual ~PMI_PiezoresistanceFactor_Base ();  
  
    virtual bool IsPrefactor () = 0;  
  
    virtual void compute (const Input& input, Output& output) = 0;  
};
```

The prototype for the virtual constructor is given as:

```
typedef PMI_PiezoresistanceFactor_Base*  
new_PMI_PiezoresistanceFactor_Base_func (const  
PMI_Environment& env);  
extern "C" new_PMI_PiezoresistanceFactor_Base_func  
new_PMI_ePiezoresistanceFactor_Base;  
extern "C" new_PMI_PiezoresistanceFactor_Base_func  
new_PMI_hPiezoresistanceFactor_Base;
```

---

## Stress

Sentaurus Device supports a PMI for mechanical stress (see [Chapter 31 on page 935](#)). The name of the PMI model must appear in the `Piezo` section of the command file:

```
Physics {  
    Piezo ( Stress = pmi_model_name )  
}
```

## Dependencies

A PMI stress model has no explicit dependencies. However, it can depend on doping concentrations and mole fractions through the runtime support.

## Chapter 39: Physical Model Interface

### Mechanical Stress

The PMI model must compute the following results:

```
stress_xx    XX component of stress tensor [Pa]
stress_yy    YY component of stress tensor [Pa]
stress_zz    ZZ component of stress tensor [Pa]
stress_yz    YZ component of stress tensor [Pa]
stress_xz    XZ component of stress tensor [Pa]
stress_xy    XY component of stress tensor [Pa]
```

## Standard C++ Interface

The following base class is declared in the file `PMIModels.h`:

```
class PMI_Stress : public PMI_Vertex_Interface {
public:
    PMI_Stress (const PMI_Environment& env);
    virtual ~PMI_Stress () ;

    virtual void Compute_StressXX
        (double& stress_xx) = 0;

    virtual void Compute_StressYY
        (double& stress_yy) = 0;

    virtual void Compute_StressZZ
        (double& stress_zz) = 0;

    virtual void Compute_StressYZ
        (double& stress_yz) = 0;

    virtual void Compute_StressXZ
        (double& stress_xz) = 0;

    virtual void Compute_StressXY
        (double& stress_xy) = 0;
};
```

The prototype for the virtual constructor is given as:

```
typedef PMI_Stress* new_PMI_Stress_func
    (const PMI_Environment& env);
extern "C" new_PMI_Stress_func new_PMI_Stress;
```

## Chapter 39: Physical Model Interface

### Mechanical Stress

## Simplified C++ Interface

The following base class is declared in the file `PMI.h`:

```
class PMI_Stress_Base : public PMI_Vertex_Base {

public:
    class Input : public PMI_Vertex_Input_Base {
public:
    };

    class Output {
public:
        pmi_float stress_xx; // xx component of stress
        pmi_float stress_yy; // yy component of stress
        pmi_float stress_zz; // zz component of stress
        pmi_float stress_yz; // yz component of stress
        pmi_float stress_xz; // xz component of stress
        pmi_float stress_xy; // xy component of stress
    };

    PMI_Stress_Base (const PMI_Environment& env);
    virtual ~PMI_Stress_Base ();

    virtual void compute (const Input& input, Output& output) = 0;
};
```

The prototype for the virtual constructor is given as:

```
typedef PMI_Stress_Base* new_PMI_Stress_Base_func
    (const PMI_Environment& env);
extern "C" new_PMI_Stress_Base_func new_PMI_Stress_Base;
```

## Example: Constant Stress

The following code returns constant values for the stress tensor:

```
#include "PMIModels.h"

class Constant_Stress : public PMI_Stress {

private:
    double xx, yy, zz, yz, xz, xy;

public:
    Constant_Stress (const PMI_Environment& env);

    ~Constant_Stress ();

    void Compute_StressXX (double& stress_xx);
    void Compute_StressYY (double& stress_yy);
    void Compute_StressZZ (double& stress_zz);
    void Compute_StressYZ (double& stress_yz);
```

## Chapter 39: Physical Model Interface

### Mechanical Stress

```
void Compute_StressXZ (double& stress_xz);
void Compute_StressXY (double& stress_xy);

};

Constant_Stress::
Constant_Stress (const PMI_Environment& env) :
    PMI_Stress (env)
{ xx = InitParameter ("xx", 100);
  yy = InitParameter ("yy", -4e9);
  zz = InitParameter ("zz", 300);
  yz = InitParameter ("yz", 400);
  xz = InitParameter ("xz", 500);
  xy = InitParameter ("xy", 600);
}

Constant_Stress::
~Constant_Stress ()
{
}

void Constant_Stress::
Compute_StressXX (double& stress_xx)
{ stress_xx = xx;
}

void Constant_Stress::
Compute_StressYY (double& stress_yy)
{ stress_yy = yy;
}

void Constant_Stress::
Compute_StressZZ (double& stress_zz)
{ stress_zz = zz;
}

void Constant_Stress::
Compute_StressYZ (double& stress_yz)
{ stress_yz = yz;
}

void Constant_Stress::
Compute_StressXZ (double& stress_xz)
{ stress_xz = xz;
}

void Constant_Stress::
Compute_StressXY (double& stress_xy)
{ stress_xy = xy;
}

extern "C"
PMI_Stress* new_PMI_Stress (const PMI_Environment& env)
```

## Chapter 39: Physical Model Interface

### Traps and Fixed Charges

```
{ return new Constant_Stress (env); }
```

---

## Traps and Fixed Charges

This section presents the following PMIs:

- [Trap Capture and Emission Rates](#)
- [Trap Energy Shift](#)

---

### Trap Capture and Emission Rates

This PMI model can be used to define either capture and emission rates for traps (see  $c_C^n$ ,  $c_V^p$ ,  $e_C^n$ , and  $e_V^p$  in [Trap Occupation Dynamics on page 553](#)) or the transitions between states of multistate configurations (see [Specifying Multistate Configurations on page 586](#)). The model is an arbitrary function of  $n$ ,  $p$ ,  $T$ ,  $T_n$ ,  $T_p$ , and  $F$ .

### Traps

In the command file, specify the model with the `CBRate` and `VBRate` options to `Traps`. For example:

```
Traps( (CBRate=( "modelX" ,17) VBRate="modelY" ) ... )
```

uses the user-specified model `modelX` to compute  $c_C^n$  and  $e_C^n$  (see [Local Capture and Emission Rates on page 555](#)), and `modelY` to compute  $c_V^p$  and  $e_V^p$ . The `17` is passed as the second argument to the virtual constructor for `modelX`; `modelY` is passed as a default value of `0` instead. The interpretation of these integers is left to the user-specified models. They allow you to select among different parameters or model variants, without having to reimplement the full model for each different choice of parameters or each minor model variation.

### Multistate Configurations

The present model is used for transitions of multistate configurations by the keyword `CEModel` (see [Specifying Multistate Configurations on page 586](#)). For example:

```
MSConfig { ...
    Transition ( Name="t1" To = "c" From="a" CEModel ( "modelX" 5 ) )
}
```

where `5` is the optional model index parameter that defaults to `0`.

## Chapter 39: Physical Model Interface

### Traps and Fixed Charges

## Dependencies

The capture and emission rates can depend on the variables:

n	Electron density [cm <sup>-3</sup> ]
p	Hole density [cm <sup>-3</sup> ]
t	Lattice temperature [K]
t <sub>n</sub>	Electron temperature [K]
t <sub>p</sub>	Hole temperature [K]
f	Electric field [Vcm <sup>-1</sup> ]

The PMI model must compute the following results (note that the carrier type that is captured and emitted is determined by the band to which the model is applied):

capture      Capture rate [s<sup>-1</sup>]

emission      Emission rate [s<sup>-1</sup>]

In the case of the standard interface, the following derivatives must be computed as well:

d<sub>capture</sub>dn      Derivative of capture rate with respect to n [s<sup>-1</sup>cm<sup>3</sup>]

d<sub>emission</sub>dn      Derivative of emission rate with respect to n [s<sup>-1</sup>cm<sup>3</sup>]

d<sub>capture</sub>dp      Derivative of capture rate with respect to p [s<sup>-1</sup>cm<sup>3</sup>]

d<sub>emission</sub>dp      Derivative of emission rate with respect to p [s<sup>-1</sup>cm<sup>3</sup>]

d<sub>capture</sub>dt      Derivative of capture rate with respect to t [s<sup>-1</sup>K<sup>-1</sup>]

d<sub>emission</sub>dt      Derivative of emission rate with respect to t [s<sup>-1</sup>K<sup>-1</sup>]

d<sub>capture</sub>d<sub>t<sub>n</sub></sub>      Derivative of capture rate with respect to t<sub>n</sub> [s<sup>-1</sup>K<sup>-1</sup>]

d<sub>emission</sub>d<sub>t<sub>n</sub></sub>      Derivative of emission rate with respect to t<sub>n</sub> [s<sup>-1</sup>K<sup>-1</sup>]

## Chapter 39: Physical Model Interface

### Traps and Fixed Charges

dcapturedtp Derivative of capture rate with respect to  $t_p$  [ $s^{-1}K^{-1}$ ]

demissiondtp Derivative of emission rate with respect to  $t_p$  [ $s^{-1}K^{-1}$ ]

dcapturedf Derivative of capture rate with respect to  $f$  [ $s^{-1}V^{-1}cm$ ]

demissiondf Derivative of emission rate with respect to  $f$  [ $s^{-1}V^{-1}cm$ ]

## Standard C++ Interface

The following base class is declared in `PMIModels.h`:

```
class PMI_TrapCaptureEmission : public PMI_Vertex_Interface {
public:
    PMI_TrapCaptureEmission(const PMI_Environment& env);
    virtual ~PMI_TrapCaptureEmission();

    virtual void Compute_rates
        (const double n, const double p, const double t,
         const double tn, const double tp, const double f,
         double& capture, double& emission) = 0;

    virtual void Compute_dratesdn
        (const double n, const double p, const double t,
         const double tn, const double tp, const double f,
         double& dcapturedn, double& demissiondn) = 0;

    virtual void Compute_dratesdp
        (const double n, const double p, const double t,
         const double tn, const double tp, const double f,
         double& dcapturedp, double& demissiondp) = 0;

    virtual void Compute_dratesdt
        (const double n, const double p, const double t,
         const double tn, const double tp, const double f,
         double& dcapturedt, double& demissiondt) = 0;

    virtual void Compute_dratesdtin
        (const double n, const double p, const double t,
         const double tn, const double tp, const double f,
         double& dcapturedtn, double& demissiondtin) = 0;

    virtual void Compute_dratesdtp
        (const double n, const double p, const double t,
         const double tn, const double tp, const double f,
         double& dcapturedtp, double& demissiondtp) = 0;

    virtual void Compute_dratesdf
        (const double n, const double p, const double t,
```

## Chapter 39: Physical Model Interface

### Traps and Fixed Charges

```
    const double tn, const double tp, const double f,
    double& dcapturedf, double& demissiondf) = 0;
};
```

The following virtual constructor must be implemented:

```
typedef PMI_TrapCaptureEmission* new_PMI_TrapCaptureEmission_func
    (const PMI_Environment& env, int id);
extern "C" new_PMI_TrapCaptureEmission_func
    new_PMI_TrapCaptureEmission;
```

## Simplified C++ Interface

The following base class is declared in the file `PMI.h`:

```
class PMI_TrapCaptureEmission_Base : public PMI_Vertex_Base {

public:
    class Input : public PMI_Vertex_Input_Base {
public:
    pmi_float n;      // electron density
    pmi_float p;      // hole density
    pmi_float t;      // lattice temperature
    pmi_float tn;     // electron temperature
    pmi_float tp;     // hole temperature
    pmi_float f;      // absolute value of electric field
    pmi_float nc;     // lattice effective state density for electrons
    pmi_float nv;     // lattice effective state density for holes
    pmi_float egeff;  // effective band gap
};

    class Output {
public:
    pmi_float capture; // capture rate
    pmi_float emission; // emission rate
};

    PMI_TrapCaptureEmission_Base (const PMI_Environment& env);
    virtual ~PMI_TrapCaptureEmission_Base ();

    virtual void compute (const Input& input, Output& output) = 0;
};
```

The prototype for the virtual constructor is given as:

```
typedef PMI_TrapCaptureEmission_Base*
    new_PMI_TrapCaptureEmission_Base_func
    (const PMI_Environment& env, int id);
extern "C" new_PMI_TrapCaptureEmission_Base_func
    new_PMI_TrapCaptureEmission_Base;
```

## Chapter 39: Physical Model Interface

### Traps and Fixed Charges

## Example: Arrhenius Law

The model has the structure of the Arrhenius law. It depends on the temperature.

Table 191 Model parameters

Symbol	Parameter name	Default	Unit	Description
$\delta E$	DeltaE	0.	eV	Energy difference
$g$	g	1.	1	Degeneracy factor
$r_0$	r0	1.	s <sup>-1</sup>	Maximal transition rate
$E_{act}$	Eact	0.	eV	Activation energy

Let  $\delta E$  and  $E_{act}$  be the energy difference and activation energy. The capture and emission rates are given by:

$$c = r_0 \exp(-E_{act}/kT) \quad (1350)$$

$$e = r_0 g \exp(-E_{act} + \delta E / kT) \quad (1351)$$

The model is shipped with Sentaurus Device. A more flexible and general way to specify Arrhenius law transitions is provided by the transition model `pmi_ce_msc` (see [Arrhenius Law \(Formula=0\) on page 590](#)).

---

## Trap Energy Shift

The present PMI determines a shift  $E_{shift}$  of trap energies that depends on the electric field and the lattice temperature at the location of the vertex or at a position given with `ReferencePoint` (see [Energetic and Spatial Distribution of Traps on page 544](#)).

## Command File

The energy shift model is specified by `EnergyShift=<model name>` or `EnergyShift=(<model name>, <int>)` as an option to `Traps` in the `Physics` section. The optional integer defaults to zero and is passed as the argument `id` to the virtual constructor of the PMI (see below). The interpretation of `id` is dependent on the user-specified model.

## Dependencies

The trap energy shift can depend on the variables:

- f Electric field vector [Vcm<sup>-1</sup>], a vector with up to three components, for the field in the x-, y-, and z-direction

## Chapter 39: Physical Model Interface

### Traps and Fixed Charges

$t$  Lattice temperature [K]

The PMI model must compute the following results:

$\text{shift}$  Energy shift [eV]

In the case of the standard interface, the following derivatives must be computed as well:

$d\text{shift}/df$  Derivative of energy shift with respect to each component of  $f$  [ $\text{eVcmV}^{-1}$ ], a vector with up to three entries

$d\text{shift}/dt$  Derivative of energy shift with respect to  $t$  [ $\text{eVK}^{-1}$ ]

## Standard C++ Interface

The following base class is declared in `PMIModels.h`:

```
class PMI_TrapEnergyShift : public PMI_Vertex_Interface {  
  
public:  
    PMI_TrapEnergyShift(const PMI_Environment& env);  
    virtual ~PMI_TrapEnergyShift();  
  
    virtual void Compute_shift(  
        const double f[3], const double t, double& shift) = 0;  
    virtual void Compute_dshiftdf(  
        const double f[3], const double t, double df[3]) = 0;  
    virtual void Compute_dshiftdt(  
        const double f[3], const double t, double& dt) = 0;  
};
```

The following virtual constructor must be implemented:

```
typedef PMI_TrapEnergyShift* new_PMI_TrapEnergyShift_func  
(const PMI_Environment& env, int id);  
extern "C" new_PMI_TrapEnergyShift_func new_PMI_TrapEnergyShift;
```

## Simplified C++ Interface

The following base class is declared in the file `PMI.h`:

```
class PMI_TrapEnergyShift_Base : public PMI_Vertex_Base {  
  
public:  
    class Input : public PMI_Vertex_Input_Base {  
    public:  
        pmi_float f[3]; // electric field vector
```

## Chapter 39: Physical Model Interface

### Degradation

```
    pmi_float t;      // lattice temperature
};

class Output {
public:
    pmi_float shift; // trap energy shift
};

PMI_TrapEnergyShift_Base (const PMI_Environment& env);
virtual ~PMI_TrapEnergyShift_Base ();

virtual void compute (const Input& input, Output& output) = 0;
};
```

The prototype for the virtual constructor is given as:

```
typedef PMI_TrapEnergyShift_Base* new_PMI_TrapEnergyShift_Base_func
(const PMI_Environment& env, int id);
extern "C" new_PMI_TrapEnergyShift_Base_func
new_PMI_TrapEnergyShift_Base;
```

---

## Degradation

This section presents the following PMIs:

- [Diffusivity](#)
- [eNMP Transition Rates](#)

---

## Diffusivity

The diffusivity  $D_i \exp(-E_{di}/(kT))$  and the prefactor of the thermal diffusion term  $\alpha_{td}$  can be defined by a PMI (see [Hydrogen Transport on page 612](#)).

The name of the PMI can be specified regionwise, materialwise, or globally in the `Physics` section of the command file as follows:

```
Physics ( Material = "Oxide" ) {
    HydrogenDiffusion(
        HydrogenAtom (
            Diffusivity = pmi_model
            Alpha = pmi_otherModel
        )
        ...
    )
}
```

## Dependencies

The diffusivity and the prefactor of the thermal diffusion term can depend on the following variables:

- f Modulus of electric field [ $\text{Vcm}^{-1}$ ]
- h Density of hydrogen species (atom, molecule, ion) [ $\text{cm}^{-3}$ ]
- t Lattice temperature [K]

The PMI model must compute the following result:

- d Diffusivity  $D_i \exp(-E_{di}/(kT))$  [ $\text{cm}^2\text{s}^{-1}$ ] (PMI model for `Diffusivity`) or prefactor of thermal diffusion term  $\alpha_{td}$  [1] (PMI model for `Alpha`)

## Simplified C++ Interface

The following base class is declared in the file `PMI.h`:

```
class PMI_HydrogenDiffusivity_Base : public PMI_Vertex_Base {  
  
public:  
    const PMI_HydrogenType hydrogen_type;  
    class Input : public PMI_Vertex_Input_Base {  
public:  
        Input (const PMI_HydrogenDiffusivity_Base*  
               hydrogendiffusivity_base, const int vertex);  
        pmi_float h; // density of hydrogen atoms, molecules, or ions  
                     // (depending on type) [/cm^3]  
        pmi_float t; // lattice temperature [K]  
        pmi_float f; // absolute value of electric field [V/cm]  
    };  
  
    class Output {  
public:  
        pmi_float d; // diffusivity [cm^2*s^-1] or prefactor of the  
                     // thermal diffusion term [1]  
    };  
  
    PMI_HydrogenDiffusivity_Base (const PMI_Environment& env, const  
                                PMI_HydrogenType type);  
    virtual ~PMI_HydrogenDiffusivity_Base ();  
  
    virtual void compute (const Input& input, Output& output) = 0;  
}
```

## Chapter 39: Physical Model Interface

### Degradation

The following virtual constructor must be implemented:

```
extern "C"
PMI_HydrogenDiffusivity_Base* new_PMI_HydrogenDiffusivity_Base
(const PMI_Environment& env, const PMI_HydrogenType type);
```

## Example: Field-Dependent Hydrogen Diffusivity

The following C++ code implements a field-dependent hydrogen diffusivity:

```
#include "PMIModels.h"
class pmi_HydrogenDiffusivity : public PMI_HydrogenDiffusivity_Base {
    private:
        double d0; // [cm^2*s^-1]
        double f0; // [eV]
        double fe; // [1]
        double ed; // [eV]

    public:
        pmi_HydrogenDiffusivity (const PMI_Environment& env,
                               const PMI_HydrogenType type);
        ~pmi_HydrogenDiffusivity ();

        void compute (const Input& input, Output& output);
    };

    pmi_HydrogenDiffusivity::pmi_HydrogenDiffusivity (const PMI_Environment& env,
                                                   const PMI_HydrogenType type) :
    PMI_HydrogenDiffusivity_Base (env, type)
    {
        // for hydrogen atom
        double d0_a = InitParameter ("d0_a", 1.0e-5);
        double f0_a = InitParameter ("f0_a", 1.0e-5);
        double fe_a = InitParameter ("fe_a", 0.5);
        double ed_a = InitParameter ("ed_a", 0.5);

        // for hydrogen molecule
        double d0_m = InitParameter ("d0_m", 1.0e-5);
        double f0_m = InitParameter ("f0_m", 1.0e-5);
        double fe_m = InitParameter ("fe_m", 0.5);
        double ed_m = InitParameter ("ed_m", 0.5);

        // for hydrogen ion
        double d0_i = InitParameter ("d0_i", 1.0e-5);
        double f0_i = InitParameter ("f0_i", 1.0e-5);
        double fe_i = InitParameter ("fe_i", 0.5);
        double ed_i = InitParameter ("ed_i", 0.5);

        if (hydrogen_type == PMI_HydrogenAtom)
        {
            d0 = d0_a;
            f0 = f0_a;
```

## Chapter 39: Physical Model Interface

### Degradation

```
    fe = fe_a;
    ed = ed_a;
}
else if(hydrogen_type == PMI_HydrogenMolecule)
{
    d0 = d0_m;
    f0 = f0_m;
    fe = fe_m;
    ed = ed_m;
}
else if(hydrogen_type == PMI_HydrogenIon)
{
    d0 = d0_i;
    f0 = f0_i;
    fe = fe_i;
    ed = ed_i;
}
else
{
    std::cout << "unexpected hydrogen type!\n";
    exit(-1);
}
pmi_HydrogenDiffusivity::
~pmi_HydrogenDiffusivity ()
{
}

void pmi_HydrogenDiffusivity::
compute (const Input& input, Output& output)
{
    pmi_float kt = 0.0258519952664849131 * (input.t/300.0); // [eV]
    // field-induced activation energy lowering
    pmi_float ea = ed - f0*pow(input.f, fe); // ed - f0*(f/(1[V/cm]))^fe

    // set activation energy equal to zero if it is negative
    if(ea<0.0) ea = 0.0;
    ea /= kt;

    output.d = d0*exp(-ea);
}

extern "C"
PMI_HydrogenDiffusivity_Base* new_PMI_HydrogenDiffusivity_Base
(const PMI_Environment& env, const PMI_HydrogenType type)
{
    return new pmi_HydrogenDiffusivity (env, type);
}
```

## eNMP Transition Rates

Transition rates used in the extended nonradiative multiphonon (eNMP) model can be computed by an eNMP transition rates PMI model. The computed transition rates are used in the solution of the eNMP model rate equations (see [eNMP Model Description on page 626](#)).

To use a PMI for calculating the transition rates, specify the PMI model name as an option to `eNMP` in the Physics section of the command file:

```
Physics (MaterialInterface | RegionInterface = "<name1>/<name2>") {
    eNMP (
        pmi_model_name [StateCharge=<1 or -1>] ...
    )
}
```

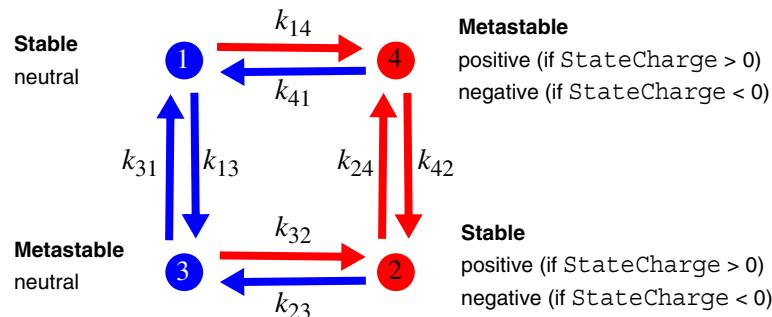
The eNMP model requires the calculation of eight transition rates occurring between four states. For the built-in eNMP model, the metastable states are designated with primes (that is,  $1'$  and  $2'$ ). To simplify the naming of the transition rates for the PMI, the following definitions are made:

- state 3 = state  $1'$
- state 4 = state  $2'$

In addition, when a PMI model is used to calculate the transition rates, the sign of the charge states can be specified with the `StateCharge` parameter in the command file. By default, the charge states are assumed to be positive (`StateCharge > 0`). To designate that the charge states are negative, specify `StateCharge < 0`.

With these definitions, the transition rates that must be calculated by the PMI model are shown in [Figure 99](#).

*Figure 99 Transition rates*



## Distinction Between Electron and Hole Transitions

For transitions involving a change in charge ( $1 \leftrightarrow 4$  and  $2 \leftrightarrow 3$ ), the PMI requires a distinction between transitions that involve holes (hole capture and hole emission) and electrons (electron capture and electron emission). This distinction is necessary so that Sentaurus Device correctly accounts for changes in carrier recombination when the eNMP model is used.

The component of a transition rate involving holes is designated with an “h” suffix, and an “e” suffix designates the electron component:

$$k_{14} = k_{14h} + k_{14e} \quad (1352)$$

$$k_{41} = k_{41h} + k_{41e} \quad (1353)$$

$$k_{23} = k_{23h} + k_{23e} \quad (1354)$$

$$k_{32} = k_{32h} + k_{32e} \quad (1355)$$

The transition rate components have the meanings shown in [Table 192](#) depending on StateCharge.

*Table 192      Definition of eNMP transition rate depending on StateCharge*

Rates	StateCharge	StateCharge
$k_{14h}, k_{32h}$	Hole capture	Hole emission
$k_{14e}, k_{32e}$	Electron	Electron
$k_{41h}, k_{23h}$	Hole emission	Hole capture
$k_{41e}, k_{23e}$	Electron	Electron

It is not required to compute both electron and hole transition components in the PMI. For example, the built-in eNMP model includes only hole transitions, and electron transitions are ignored.

## Transition Rates for All Sample Defects

Since the eNMP model uses a random sampling technique to obtain the average behavior of multiple defects, the eNMP transition rates PMI model must calculate transition rates for all of the sample defects at each vertex. For this purpose, the following parameters are passed automatically to the eNMP transition rates PMI and can be accessed using the runtime support function `InitParameter()`:

```
NumberOfSamples = InitParameter( "NumberOfSamples" , 1000 );
NumberOfVertices = InitParameter( "NumberOfVertices" , 1 );
```

## Chapter 39: Physical Model Interface

### Degradation

The `NumberOfSamples` keyword represents the value specified in the `eNMP` section of the command file. The `NumberOfVertices` keyword represents the number of interface vertices at the interface where the `eNMP` model is being used.

## Parameter Randomization

Since transition rates must be obtained for all of the sample defects, it is likely that some model parameters will need to be randomized (either using a uniform distribution or a Gaussian distribution). In this case, the randomized parameter values should be obtained and stored in the constructor of the `eNMP` transition rates PMI for *all* `NumberOfSamples` defects *and* for all `NumberOfVertices` interface vertices. Therefore, a total of `NumberOfSamples × NumberOfVertices` values should be obtained for each parameter to be randomized. The requirement to obtain randomized parameter values for all interface vertices in the PMI constructor is to ensure that each vertex uses *different* `NumberOfSamples` values for each randomized parameter.

The `eNMP` transition rates PMI includes two member functions to assist in randomizing parameter values:

- `double UnifRanNum(int &idum)` – returns a random number between 0 and 1, exclusive of the endpoints.
- `double GausRanNum(int &idum)` – returns a random number that follows a normal distribution with zero mean and unit variance.

These functions must be initialized with `idum < 0` and, thereafter, `idum` must not be changed during the sequence. For example, the following code fragment shows how two parameters can be randomized and stored for later use:

```
std::vector<double> Eti;
std::vector<double> xti;
...
NumberOfSamples = InitParameter("NumberOfSamples", 1000);
NumberOfVertices = InitParameter("NumberOfVertices", 1);
double Et0    = InitParameter("Et", -0.5);
double Etsig = InitParameter("Etsig", 0.1);
double xtmin = InitParameter("xtmin", 0.0);
double xtmax = InitParameter("xtmax", 0.0);
...
int idum = -1;
for (int i = 0; i < NumberOfSamples*NumberOfVertices; ++i) {
    double Et = Et0 + Etsig*GausRanNum(idum);
    double xt = xtmin + (xtmax - xtmin)*UnifRanNum(idum);
    ...
    Eti.push_back(Et);
    xti.push_back(xt);
    ...
}
```

## Chapter 39: Physical Model Interface

### Degradation

## Dependencies

The eNMP transition rates PMI can depend on the interface values of the following variables:

n	Electron density [cm <sup>-3</sup> ]
p	Hole density [cm <sup>-3</sup> ]
N <sub>c</sub>	Conduction band density-of-states [cm <sup>-3</sup> ]
N <sub>v</sub>	Valence band density-of-states [cm <sup>-3</sup> ]
E <sub>g</sub>	Band gap [eV]
T	Lattice temperature [K]
F	Magnitude of insulator electric field [Vcm <sup>-1</sup> ]

The PMI model must compute the following results:

k[0] <sub>i</sub> = (k <sub>14h</sub> ) <sub>i</sub>	Hole component of transition rates from state 1 to state 4 [s <sup>-1</sup> ]
k[1] <sub>i</sub> = (k <sub>14e</sub> ) <sub>i</sub>	Electron component of transition rates from state 1 to state 4 [s <sup>-1</sup> ]
k[2] <sub>i</sub> = (k <sub>41h</sub> ) <sub>i</sub>	Hole component of transition rates from state 4 to state 1 [s <sup>-1</sup> ]
k[3] <sub>i</sub> = (k <sub>41e</sub> ) <sub>i</sub>	Electron component of transition rates from state 4 to state 1 [s <sup>-1</sup> ]
k[4] <sub>i</sub> = (k <sub>23h</sub> ) <sub>i</sub>	Hole component of transition rates from state 2 to state 3 [s <sup>-1</sup> ]
k[5] <sub>i</sub> = (k <sub>23e</sub> ) <sub>i</sub>	Electron component of transition rates from state 2 to state 3 [s <sup>-1</sup> ]
k[6] <sub>i</sub> = (k <sub>32h</sub> ) <sub>i</sub>	Hole component of transition rates from state 3 to state 2 [s <sup>-1</sup> ]
k[7] <sub>i</sub> = (k <sub>32e</sub> ) <sub>i</sub>	Electron component of transition rates from state 3 to state 2 [s <sup>-1</sup> ]
k[8] <sub>i</sub> = (k <sub>13</sub> ) <sub>i</sub>	Transition rates from state 1 to state 3 [s <sup>-1</sup> ]
k[9] <sub>i</sub> = (k <sub>31</sub> ) <sub>i</sub>	Transition rates from state 3 to state 1 [s <sup>-1</sup> ]

## Chapter 39: Physical Model Interface

### Degradation

$$k[10]_i = (k_{24})_i \quad \text{Transition rates from state 2 to state 4 [s}^{-1}\text{]}$$

$$k[11]_i = (k_{42})_i \quad \text{Transition rates from state 4 to state 2 [s}^{-1}\text{]}$$

#### Note:

The subscript  $i$  in the above transition rates is shorthand that indicates that the transition rate must be computed for all `NumberOfSamples` sample defects at the interface vertex. For example:

$$k[0]_i \rightarrow k[0]_0, k[0]_1, k[0]_2, \dots, k[0]_{\text{NumberOfSamples}-1} \quad (1356)$$

In the case of the standard PMI interface, the following derivatives must also be computed:

$dkdn[0]_i, dkdn[1]_i, \dots, dkdn[11]_i$	Derivatives of all transition rates with respect to $n$ [cm $^3$ s $^{-1}$ ]
$dkdp[0]_i, dkdp[1]_i, \dots, dkdp[11]_i$	Derivatives of all transition rates with respect to $p$ [cm $^3$ s $^{-1}$ ]
$dkdNc[0]_i, dkdNc[1]_i, \dots, dkdNc[11]_i$	Derivatives of all transition rates with respect to $N_c$ [cm $^3$ s $^{-1}$ ]
$dkdNv[0]_i, dkdNv[1]_i, \dots, dkdNv[11]_i$	Derivatives of all transition rates with respect to $N_v$ [cm $^3$ s $^{-1}$ ]
$dkdEg[0]_i, dkdEg[1]_i, \dots, dkdEg[11]_i$	Derivatives of all transition rates with respect to $E_g$ [eV $^{-1}$ s $^{-1}$ ]
$dkdT[0]_i, dkdT[1]_i, \dots, dkdT[11]_i$	Derivatives of all transition rates with respect to $T$ [K $^{-1}$ s $^{-1}$ ]
$dkdF[0]_i, dkdF[1]_i, \dots, dkdF[11]_i$	Derivatives of all transition rates with respect to $F$ [cmV $^{-1}$ s $^{-1}$ ]

## Chapter 39: Physical Model Interface

### Degradation

#### Note:

Similar to the transition rates, the subscript  $i$  in the above derivatives is shorthand that indicates that the derivatives must be computed for all NumberOfSamples sample defects at the interface vertex. For example:

$$\text{dkdn}[0]_i \rightarrow \text{dkdn}[0]_0, \text{dkdn}[0]_1, \text{dkdn}[0]_2, \dots, \text{dkdn}[0]_{\text{NumberOfSamples} - 1} \quad (1357)$$

The derivative array indices correspond to the array indices for the transition rates. For example:

$$\text{dkdn}[0]_i = \frac{d(k_{14h})_i}{dn}, \text{dkdn}[1]_i = \frac{d(k_{14e})_i}{dn}, \dots \quad (1358)$$

## Standard C++ Interface

The following base class is declared in the file `PMIModels.h`:

```
class PMI_eNMPTransitionRates : public PMI_Vertex_Interface {

public:
    class Input {
public:
    double n;      // electron density
    double p;      // hole density
    double Nc;     // conduction-band effective density-of-states
    double Nv;     // valence-band effective density-of-states
    double Eg;     // bandgap
    double T;      // lattice temperature
    double F;      // insulator electric field
    int vindex;   // vertex index (0 ≤ vindex ≤ NumberOfVertices - 1 )
};

    class Output {
public:
    std::vector<double> k[12];      // transition rates
    std::vector<double> dkdn[12];    // derivatives wrt electron
                                      // density
    std::vector<double> dkdp[12];    // derivatives wrt hole density
    std::vector<double> dkdNc[12];   // derivatives wrt to conduction
                                      // band DOS
    std::vector<double> dkdNv[12];   // derivatives wrt to valence
                                      // band DOS
    std::vector<double> dkdEg[12];   // derivatives wrt bandgap
    std::vector<double> dkdT[12];    // derivatives wrt lattice
                                      // temperature
    std::vector<double> dkdF[12];    // derivatives wrt insulator
                                      // electric field
};

    PMI_eNMPTransitionRates (const PMI_Environment& env);
    virtual ~PMI_eNMPTransitionRates ();
};
```

## Chapter 39: Physical Model Interface

### Degradation

```
// method to be implemented by user
virtual void compute(const Input& input, Output& output) = 0;

// random number generators
double UnifRanNum (int &idum);
double GausRanNum (int &idum);
};
```

The following virtual constructor must be implemented:

```
typedef PMI_eNMPTransitionRates*
new_PMI_eNMPTransitionRates_func (const PMI_Environment& env);
extern "C" new_PMI_eNMPTransitionRates_func
new_PMI_eNMPTransitionRates;
```

## Simplified C++ Interface

The following base class is declared in the file `PMI.h`:

```
class PMI_eNMPTransitionRates_Base : public PMI_Vertex_Base {

public:
    class Input : public PMI_Vertex_Input_Base {
public:
    Input (const PMI_eNMPTransitionRates_Base* enmprates_base,
    const int vertex);
    pmi_float n; // electron density
    pmi_float p; // hole density
    pmi_float Nc; // conduction-band effective density-of-states
    pmi_float Nv; // valence-band effective density-of-states
    pmi_float Eg; // bandgap
    pmi_float T; // lattice temperature
    pmi_float F; // insulator electric field
    int vindex; // vertex index (0≤vindex≤NumberOfVertices - 1 )
};

    class Output {
public:
    std::vector<pmi_float> k[12]; // transition rates
};

    PMI_eNMPTransitionRates_Base (const PMI_Environment& env);

    virtual ~PMI_eNMPTransitionRates_Base ();

    virtual void compute(const Input& input, Output& output) = 0;

    // random number generators
    double UnifRanNum (int &idum);
    double GausRanNum (int &idum);
};
```

## Chapter 39: Physical Model Interface

### Degradation

The prototype for the virtual constructor is given as:

```
typedef PMI_eNMPTransitionRates_Base*
new_PMI_eNMPTransitionRates_Base_func
(const PMI_Environment& env);
extern "C" new_PMI_eNMPTransitionRates_Base_func
new_PMI_eNMPTransitionRates_Base;
```

## Example: eNMP Model Transition Rates

This example illustrates the implementation of the eNMP model transition rates using the standard C++ interface:

```
#include "PMI.h"
#include <cmath>
#include <vector>

// Implementation of transitions rates for the eNMP model.

class eNMPRates : public PMI_eNMPTransitionRates {

protected:
    int NumberOfSamples, NumberOfVertices;
    double Xsec, Vth, x0, nu0;
    std::vector<double> Eti;
    std::vector<double> Etpi;
    std::vector<double> Ri;
    std::vector<double> Rpi;
    std::vector<double> ESi; Model
    std::vector<double> ESp;
    std::vector<double> ET2pi;
    std::vector<double> E1p1i;
    std::vector<double> E2p2i;
    std::vector<double> xti;

public:
    eNMPRates (const PMI_Environment& env);
    ~eNMPRates ();
    void compute (const Input& input, Output& output);
};

eNMPRates::
eNMPRates (const PMI_Environment& env) : PMI_eNMPTransitionRates (env)
{
    NumberOfSamples = InitParameter ("NumberOfSamples", 1000); // 1
    NumberOfVertices = InitParameter ("NumberOfVertices", 1); // 1
    Xsec = InitParameter ("Xsec", 1.0e-15); // cm^2
    Vth = InitParameter ("Vth", 1.2e+07); // cm/s
    x0 = InitParameter ("x0", 0.5); // A
    nu0 = InitParameter ("nu0", 1.0e+13); // s^-1

    double Et0 = InitParameter ("Et", -0.5); // eV
    double Etp0 = InitParameter ("Etp", 0.5); // eV
```

## Chapter 39: Physical Model Interface

### Degradation

```
double R0      = InitParameter ("R"      , 0.6); // 1
double Rp0     = InitParameter ("Rp"     , 0.6); // 1
double ES0      = InitParameter ("ES"     , 1.0); // eV
double ESp0     = InitParameter ("ESp"    , 1.0); // eV
double ET2p0   = InitParameter ("ET2p"   , 0.5); // eV
double E1p10    = InitParameter ("E1p1"   , 1.0); // eV
double E2p20    = InitParameter ("E2p2"   , 0.5); // eV
double xmin     = InitParameter ("xmin"   , 0.0); // A

double Etsig    = InitParameter ("Etsig"  , 0.1); // eV
double Etpsig   = InitParameter ("Etpsig" , 0.1); // eV
double Rsig     = InitParameter ("Rsig"   , 0.1); // 1
double Rpsig    = InitParameter ("Rpsig"  , 0.1); // 1
double ESSig    = InitParameter ("ESSig"  , 0.1); // eV
double ESpSig   = InitParameter ("ESpSig" , 0.1); // eV
double ET2psig  = InitParameter ("ET2psig", 0.1); // eV
double E1plsig  = InitParameter ("E1plsig", 0.1); // eV
double E2p2sig  = InitParameter ("E2p2sig", 0.1); // eV
double xmax     = InitParameter ("xmax"   , 0.0); // A

int idum = -1;
for (int i = 0; i < NumberOfSamples*NumberOfVertices; ++i) {
    double Et    = Et0 + Etsig *GausRanNum(idum);
    double Etp   = Etp0 + Etpsig *GausRanNum(idum);
    double R     = R0 + Rsig *GausRanNum(idum);
    double Rp    = Rp0 + Rpsig *GausRanNum(idum);
    double ES    = ES0 + ESSig *GausRanNum(idum);
    double ESp   = ESp0 + ESpSig *GausRanNum(idum);
    double ET2p  = ET2p0 + ET2psig*GausRanNum(idum);
    double E1pl  = E1p10 + E1plsig*GausRanNum(idum);
    double E2p2  = E2p20 + E2p2sig*GausRanNum(idum);
    double xt = xmin + (xmax - xmin)*UnifRanNum(idum);
    if (R < 1.0e-10) R = 1.0e-10;
    if (Rp < 1.0e-10) Rp = 1.0e-10;
    if (ES < 1.0e-10) ES = 1.0e-10;
    if (ESp < 1.0e-10) ESp = 1.0e-10;
    Eti.push_back(Et);
    Etpi.push_back(Etp);
    Ri.push_back(R);
    Rpi.push_back(Rp);
    ESi.push_back(ES);
    ESpi.push_back(ESp);
    ET2pi.push_back(ET2p);
    E1pli.push_back(E1pl);
    E2p2i.push_back(E2p2);
    xti.push_back(xt);
}
}

eNMPRates::
~eNMPRates ()
{
```

## Chapter 39: Physical Model Interface

### Degradation

```
void eNMPRates::  
compute (const Input& input, Output& output)  
{  
    // Compute NBTI transition rates .  
    // The order of transition rates and derivatives are:  
    // k14h, k14e, k41h, k41e (k14 = k14h + k14e; k41 = k41h + k41e)  
    // k23h, k23e, k32h, k32e (k23 = k23h + k23e; k32 = k32h + k32e)  
    // k13, k31, k24, k42  
  
    // Get the input variables.  
    const double n = input.n;      // cm^-3  
    const double p = input.p;      // cm^-3  
    const double Nc = input.Nc;    // cm^-3  
    const double Nv = input.Nv;    // cm^-3  
    const double Eg = input.Eg;    // eV  
    const double T = input.T;     // K  
    const double F = input.F;     // V/cm (this is always |F|)  
    const double kT = 8.617331e-5*T; // kT in eV  
    const int vindex = input.vindex;  
  
    // Initialize the output variables.  
    for (int i = 0; i < 12; ++i) {  
        output.k[i].resize(NumberOfSamples, 0.0);  
        output.dkdn[i].resize(NumberOfSamples, 0.0);  
        output.dkdः[i].resize(NumberOfSamples, 0.0);  
        output.dkdNc[i].resize(NumberOfSamples, 0.0);  
        output.dkdNv[i].resize(NumberOfSamples, 0.0);  
        output.dkdEg[i].resize(NumberOfSamples, 0.0);  
        output.dkdः[i].resize(NumberOfSamples, 0.0);  
        output.dkdF[i].resize(NumberOfSamples, 0.0);  
    }  
  
    // Calculate the rates for each sample.  
    for (int is = 0; is < NumberOfSamples; ++is) {  
        int ii = is + vindex*NumberOfSamples;  
        const double Et = Eti[ii];  
        const double Etp = Etpi[ii];  
        const double R = Ri[ii];  
        const double Rp = Rpi[ii];  
        const double ES = ESi[ii];  
        const double ESP = ESp[i];  
        const double ET2p = ET2pi[ii];  
        const double E1p1 = E1pli[ii];  
        const double E2p2 = E2p2i[ii];  
        const double xt = xti[ii];  
        const double sigma = Xsec * exp(-xt/x0);  
        const double xtcः = xt*1.0e-8;  
        const double R1p1 = R + 1.0;  
        const double R3p1 = Rp + 1.0;  
        const double R1fac = R1p1*sqrt(R1p1)*sigma*Vth;  
        const double R3fac = R3p1*sqrt(R3p1)*sigma*Vth;
```

## Chapter 39: Physical Model Interface

### Degradation

```
// Trap energies relative to valence band.  
const double Etv = Et + xtcn * F;  
const double Etpv = Etp + xtcn * F;  
  
// Get transition energies and derivatives.  
const double E14 = (ES/R1p1 - R*(Etv - ET2p))/R1p1;  
const double E41 = E14 + Etv - ET2p;  
const double E32 = (ESp/R3p1 - Rp*Etpv)/R3p1;  
const double E23 = E32 + Etpv;  
const double dE14dF = -R/R1p1*xtcn;  
const double dE41dF = dE14dF + xtcn;  
const double dE32dF = -Rp/R3p1*xtcn;  
const double dE23dF = dE32dF + xtcn;  
const double E13 = E1p1 + Etp - Et;  
const double E31 = E1p1;  
const double E24 = E2p2 + ET2p;  
const double E42 = E2p2;  
  
// Get transition rates and derivatives.  
const double k14 = R1fac*p*exp(-E14/kT);  
const double dk14dp = k14/p;  
const double dk14dT = k14*(E14/kT)/T;  
const double dk14dF = k14*(-dE14dF/kT);  
const double k41 = R1fac*Nv*exp(-E41/kT);  
const double dk41dNv = k41/Nv;  
const double dk41dT = k41*(E41/kT)/T;  
const double dk41dF = k41*(-dE41dF/kT);  
const double k23 = R3fac*Nv*exp(-E23/kT);  
const double dk23dNv = k23/Nv;  
const double dk23dT = k23*(E23/kT)/T;  
const double dk23dF = k23*(-dE23dF/kT);  
const double k32 = R3fac*p*exp(-E32/kT);  
const double dk32dp = k32/p;  
const double dk32dT = k32*(E32/kT)/T;  
const double dk32dF = k32*(-dE32dF/kT);  
const double k13 = nu0*exp(-E13/kT);  
const double dk13dT = k13*(E13/kT)/T;  
const double k31 = nu0*exp(-E31/kT);  
const double dk31dT = k31*(E31/kT)/T;  
const double k24 = nu0*exp(-E24/kT);  
const double dk24dT = k24*(E24/kT)/T;  
const double k42 = nu0*exp(-E42/kT);  
const double dk42dT = k42*(E42/kT)/T;  
  
// Fill the output variables.  
output.k[0][is]=k14;  
output.dkdp[0][is]=dk14dp;  
output.dkdT[0][is]=dk14dT;  
output.dkdf[0][is]=dk14dF;  
output.k[2][is]=k41;  
output.dkdNv[2][is]=dk41dNv;  
output.dkdT[2][is]=dk41dT;  
output.dkdf[2][is]=dk41dF;
```

## Chapter 39: Physical Model Interface

### Ferroelectrics and Ferromagnetics

```
    output.k[4][is]=k23;
    output.dkdNv[4][is]=dk23dNv;
    output.dkdT[4][is]=dk23dT;
    output.dkdF[4][is]=dk23dF;
    output.k[6][is]=k32;
    output.dkdP[6][is]=dk32dp;
    output.dkdT[6][is]=dk32dT;
    output.dkdF[6][is]=dk32dF;
    output.k[8][is]=k13;
    output.dkdT[8][is]=dk13dT;
    output.k[9][is]=k31;
    output.dkdT[9][is]=dk31dT;
    output.k[10][is]=k24;
    output.dkdT[10][is]=dk24dT;
    output.k[11][is]=k42;
    output.dkdT[11][is]=dk42dT;
}
}

extern "C"
PMI_eNMPTransitionRates*
new_PMI_eNMPTransitionRates (const PMI_Environment& env)
{
    return new eNMPRates (env);
}
```

---

## Ferroelectrics and Ferromagnetics

This section presents the following PMIs:

- [Ferroelectrics](#)
- [Ferroelectrics Hysteresis](#)
- [Ferromagnetism and Spin Transport](#)

---

## Ferroelectrics

Higher order terms of polarization can be included through the PMI interface, which solves the following equation:

$$LK2 = \sum_i a_i P^i \quad (1359)$$

where  $P$  is the polarization,  $i$  is the power, and  $a_i$  is the corresponding coefficient. [Equation 1359](#) is added to the built-in Ginzburg–Landau–Khalatnikov equation, [Equation 963 on page 910](#).

You set PMIModel in the FEPolarization statement to activate the PMI as follows:

```
Physics (Region="FE1") {
    FEPolarization (
        direction="x" PMIModel=FE_pmi(power=(7,9,13...))
        coeff=(a7,a9,a13...))
}
```

The input parameters are two vectors:

- The power vector contains the powers that are used.
- The coeff are the corresponding coefficients for those powers as defined in [Equation 1359](#).

**Note:**

This PMI feature can be used only with the default version of the solver (see [Default on page 915](#)).

The following is an example of the PMI:

```
#include <vector>
#include <assert.h>
#include "PMI.h"
class FE_pmi: public PMI_FEPolarization_Base {
private:
    // user parameters
    std::vector<double> power, coeff;

public:
    FE_pmi(const PMI_Environment& env);
    ~FE_pmi();
    void compute(const Input& input, Output& output);
};

FE_pmi::FE_pmi(const PMI_Environment&
env):PMI_FEPolarization_Base(env) {
    InitParameter ("power", power);
    InitParameter ("coeff", coeff);
}

FE_pmi::~FE_pmi() {
    power.clear();
    coeff.clear();
}

void FE_pmi::compute(const Input& input, Output& output) {
    const pmi_float& p = input.p;
    pmi_float lk=0.;
    assert(power.size()==coeff.size());
    if(power.size()>0) {
        for(size_t i=0;i<power.size(); ++i) {
```

## Chapter 39: Physical Model Interface

### Ferroelectrics and Ferromagnetics

```
        lk += coeff[i]*pow(p,power[i]);
    }
}
output.LKPol = lk;
}

extern "C" PMI_FEPolarization_Base* new_PMI_FEPolarization_Base(
    const PMI_Environment& env) {
    return new FE_pmi(env);
}
```

---

## Ferroelectrics Hysteresis

The ferroelectrics model, introduced in [Ferroelectrics Model on page 908](#), can be extended by defining the auxiliary polarization,  $P_{\text{aux}}$  in [Equation 958 on page 908](#), as a general algebraic function of the auxiliary electric field  $F_{\text{aux}}$ :

$$P_{\text{aux}} = c \cdot P_s \cdot f(F_{\text{aux}} \pm F_c) + P_{\text{off}} \quad (1360)$$

where  $P_s$  is the saturation polarization of the material,  $F_c$  is the coercive field, and the function  $f$  serves as the shape function of the hysteresis. The PMI in Sentaurus Device allows you to redefine the calculation of  $f$ .  $P_{\text{off}}$  and  $c$  in [Equation 1360](#) result from the polarization history of the material (see the discussion in [Ferroelectrics Model on page 908](#)). This PMI also provides access to the relaxation times for the auxiliary field  $\tau_E$  (see [Equation 957 on page 908](#)) and for the polarization  $\tau_P$  (see [Equation 960 on page 908](#)), used in transient simulations.

In the command file, the name of the ferroelectrics model is given as an argument to the `Polarization` keyword. The following example activates the ferroelectric model `pmi_model_name` in region "R.Ferro":

```
Physics (Region="R.Ferro") {
    Polarization(pmi_model_name)
}
```

The materialwise specification of PMI models is also possible.

## Dependencies

A PMI ferroelectric hysteresis model can depend on the variable:

`f` Component of electric field vector [ $\text{Vcm}^{-1}$ ]

It can be used to compute the following results:

`shape` Polarization shape function  $f$  [1]

## Chapter 39: Physical Model Interface

### Ferroelectrics and Ferromagnetics

tauE      Relaxation time for the auxiliary field  $\tau_E$  [s]

tauP      Relaxation time for the polarization  $\tau_P$  [s]

Besides the calculation of the model output values, the following partial derivatives with respect to the input electric field variable  $f$  must be computed as well:

dshapEdf    Derivative of  $f$  with respect to  $f$  [ $V^{-1} \text{ cm}$ ]

dtauEdf    Derivative of  $\tau_E$  with respect to  $f$  [ $sV^{-1} \text{ cm}$ ]

dtauPdf    Derivative of  $\tau_P$  with respect to  $f$  [ $sV^{-1} \text{ cm}$ ]

## Standard C++ Interface

The following base class is declared in the file `PMIModels.h`:

```
class PMI_FerroPolarization : public PMI_Vertex_Common_Base {
public:
    PMI_FerroPolarization (const PMI_Environment& env); virtual
    ~PMI_FerroPolarization ();

    double ReadPolarizationParameter(char const* name, int i);

    virtual bool Use_shapE() { return false; }

    virtual void Compute_shapE
        (const int i,      const double f,      double& shapE);
    virtual void Compute_dshapEdf
        (const int i,      const double f,      double& dshapEdf);
    virtual bool Use_tauE() { return false; }

    virtual void Compute_tauE
        (const int i,      const double f,      double& tauE);
    virtual void Compute_dtauEdf
        (const int i,      const double f,      double& dtauEdf);
    virtual bool Use_tauP() { return false; }

    virtual void Compute_tauP
        (const int i,      const double f,      double& tauP);
    virtual void Compute_dtauPdf
        (const int i,      const double f,      double& dtauPdf);};
```

## Chapter 39: Physical Model Interface

### Ferroelectrics and Ferromagnetics

The prototype for the virtual constructor is:

```
typedef PMI_FerroPolarization* new_PMI_FerroPolarization_func  
(const PMI_Environment& env); Extern "C" new  
new_PMI_FerroPolarization_func new_PMI_FerroPolarization
```

By default, functions computing the results of a PMI ferroelectric hysteresis model and its derivatives are not invoked. To force Sentaurus Device to use the result computed with the `Compute_result` method, a designated `Use_result` function in the PMI must return `true`. In this case, a corresponding function `Compute_dresultdf` calculating the partial derivative of the result with respect to the electric field variable `f` is also used and must be implemented.

During a Newton iteration, Sentaurus Device computes the PMI model results and derivatives, which are declared as used for each mesh element. The functions `Compute_result` and `Compute_dresultdf` are evaluated along the main axes of the mesh coordinate system as indicated by the input argument `i`. This integer variable can take values from 0 to `d`, where `d` is equal to the dimension of the problem. The input variable `f` is the component of the electric field vector along the `i`-th axis of the mesh. The output variables are used to evaluate a PMI model along the corresponding axis.

---

## Ferromagnetism and Spin Transport

The following PMI models for ferromagnetism and spin transport are supported:

- [User-Defined Interlayer Exchange Coupling](#)
- [User-Defined Bulk or Interface Contributions to the Effective Magnetic Field](#)
- [User-Defined Magnetostatic Potential Calculation](#)

All PMI models for ferromagnetism and spin transport are based on the simplified C++ interface.

## User-Defined Interlayer Exchange Coupling

Interfaces of ferromagnetic regions separated by thin paramagnetic layers lead to an interlayer coupling energy density  $U_{\text{interlayer}}$  [5]. As usual, the derivative of this energy density with respect to the local magnetization contributes to the effective magnetic field of the LLG equation. In contrast to [Equation 970 on page 926](#),  $U_{\text{interlayer}}$  in this case is a surface energy density, not a volume energy density. The resulting effective magnetic field contribution has units of A rather than A/m; with the observation that  $(J/T)/m^2 = A$ , this is seen to correspond simply to a surface density of magnetic dipoles.

The theory of interlayer exchange is well developed (for example, [6]). It explains why interlayer exchange oscillates between ferromagnetic and anti-ferromagnetic behavior as a function of the paramagnetic spacer thickness, and it establishes a clear link between

oscillation periods present in this thickness dependency and *critical spanning vectors* of the Fermi surface of the spacer material. Despite this, the quantitative prediction of the interaction remains difficult even in ideal thin film stacks. Structural non-idealities (such as surface roughness) can cause additional complexity such as the emergence of bi-quadratic terms in the coupling energy, which favor orthogonal alignment of the magnetization directions in the ferromagnetic regions. Therefore, the most appropriate functional form for describing the coupling strength might depend on the particular use case. For this reason, it was decided to provide a generic infrastructure for assembling interlayer exchange contributions into the LLG equation, but to leave the choice of the particular expression to assemble on the *interlayer exchange edges* to users.

### Syntax of Command File and Parameter File

An interlayer exchange PMI is activated for a particular interface by adding the following line to the corresponding interface `Physics` section of the command file:

```
Magnetism(InterlayerExchange(PMImodel=<name>))
```

If the PMI model uses named model parameters (for example, by calling `InitParameter()`), then the parameters are read from the interface-specific section of the parameter file.

### Base Class for Interlayer Exchange PMIs

PMI models for the interlayer exchange terms of the LLG equation are derived from the base class `PMI_LLGInterlayerExchange_Base`, which has the following definition:

```
//! Base-class for interlayer exchange terms the LLG equation
class PMI_LLGInterlayerExchange_Base : public PMI_Device_Base {
public:
    class Input : public PMI_Device_Input_Base {
public:
    //! Constructor
    Input(const PMI_Device_Base* );
    pmi_float m_loc[3]; ///< magnetization dir. at the local end of
                        //the edge
    pmi_float m_rem[3]; ///< magnetization dir. at the remote end of
                        //the edge
    pmi_float length;   ///< the length of the interlayer exchange
                        //edge [m]
    };
    class Output {
public:
    Output(const PMI_Device_Base *base);

    //! Derivative of surface energy density w.r.t. to local
    //magnetization
    pmi_float dU_by_dmloc[3];
    };
PMI_LLGInterlayerExchange_Base (const PMI_Device_Environment& env);
```

## Chapter 39: Physical Model Interface

### Ferroelectrics and Ferromagnetics

```
virtual ~PMI_LLGInterlayerExchange_Base ();
virtual void compute (const Input& input, Output& output) = 0;
};
```

The PMI implementer must provide the function:

```
compute(const Input& input, Output& output)
```

Here, `compute()` is called once for each interlayer exchange edge, and the `input` object contains the magnetization directions at the local and remote ends of the edge (as Cartesian unit vectors) as well as the layer thickness (in meter).

The function writes the gradient of the interlayer-exchange energy density with respect to the local magnetization direction into the `dU_by_dmloc` field of the `output` object (unit: J/m<sup>2</sup>).

### Example: ILE Model With a Simple Oscillatory Thickness Dependency

This model implements an interlayer exchange (ILE) surface energy density of the form  $U_{\text{interlayer}} = J_1(t)\vec{m}_{\text{loc}} \cdot \vec{m}_{\text{rem}}$ .

For the thickness-dependent coupling strength  $J_1(t)$ , a phase-shifted sine function with a  $t^{-2}$  envelope is assumed:

$$J_1(t) = J_0 \sin \frac{2\pi t}{\Lambda} + \delta / t^2 \quad (1361)$$

Here,  $t$  (in Å) is the spacer thickness, and  $\Lambda$  (in Å) is the oscillation period. Instead of the rather inconvenient parameters  $J_0$  and  $\delta$ , users are expected to supply the coupling strength  $J_{\max}$  (in mJ/m<sup>2</sup>) at the first anti-ferromagnetic peak of  $J_1(t)$  and the thickness  $t_{\max}$  (in Å) at which this maximum occurs.

### Implementation of the Simple Interlayer Exchange PMI

To implement the simple interlayer exchange PMI model, you must declare a derived class:

```
#include "PMI.h"
#include <cmath>

class InterlayerExchange_sinD_over_D2 : public
    PMI_LLGInterlayerExchange_Base
{
public:
    InterlayerExchange_sinD_over_D2(const PMI_Device_Environment &env);
    void compute(const Input &input, Output &output);

    // Auxiliary function for thickness dependence of coupling strength
    inline pmi_float func(const pmi_float &x, const pmi_float &shift) {
        return sin(x + shift) / (x * x);
    }
private:
```

## Chapter 39: Physical Model Interface

### Ferroelectrics and Ferromagnetics

```
    double Jmax;      ///< Coupling strength at first AF peak [mJ/m^2]
    double tmax;      ///< Position of first peak [Aangstroem]
    double Lambda;    ///< Period of oscillation [Aangstroem]
    pmi_float delta; //;< Phase shift to move first AF peak to tmax
    pmi_float scale; //;< Scaling factor to scale first AF peak to Jmax
};
```

The constructor of the derived class reads the model parameters from the .par file and determines the phase shift and the prefactor in  $J_1(t)$  from  $J_{\max}$  and  $t_{\max}$ .

```
InterlayerExchange_sinD_over_D2::  
InterlayerExchange_sinD_over_D2(const PMI_Device_Environment &env)  
: PMI_LLGIterlayerExchange_Base(env)  
{  
    Jmax    = InitParameter("Jmax", 0.0);    // unit: [mJ/m^2]  
    Lambda = InitParameter("Lambda", 0.0);   // unit: [Aangstroem]  
    tmax   = InitParameter("tmax", 0.0);     // unit: [Aangstroem]  
  
    // adjust delta and scale to position first peak at (tmax, Jmax)  
    pmi_float tmax_scaled = tmax * 2 * M_PI / Lambda;  
    delta = M_PI - tmax_scaled + atan(0.5 * tmax_scaled);  
    scale = 1.0 / func(tmax_scaled, delta);  
}
```

The compute() function computes  $\vec{\nabla}_{\vec{m}_{loc}} U_{interlayer} = J_1(t) \vec{m}_{rem}$  for a single interlayer exchange edge (note the unit conversion factors):

```
compute(const Input &input, Output &output) {  
    pmi_float scaled_distance = 1e10 * input.length * 2*M_PI / Lambda;  
    pmi_float f = -1e-3 * scale * Jmax * func(scaled_distance, delta);  
    output.dU_by_dmloc[0] = f * input.m_rem[0];  
    output.dU_by_dmloc[1] = f * input.m_rem[1];  
    output.dU_by_dmloc[2] = f * input.m_rem[2];  
}
```

Finally, you must provide a so-called virtual constructor function that allocates a variable of the new class:

```
extern "C"  
PMI_LLGIterlayerExchange_Base*  
new_PMI_LLGIterlayerExchange_Base(const PMI_Device_Environment&  
env) {  
    return new InterlayerExchange_sinD_over_D2(env);  
}
```

**Note:**

This function must have C linkage and exactly the same name as declared in the `PMI.h` header file.

If `Magnetism(InterlayerExchange)` is specified in the `.cmd` file without providing a PMI model name, then Sentaurus Device loads an implementation of the above model as the default ILE model. For the purpose of reading parameters from the `.par` file, the model name `InterlayerExchange` is used.

## User-Defined Bulk or Interface Contributions to the Effective Magnetic Field

The base class `PMI_LLGHeff_Base` has been provided for assembling extra generic contributions to the effective magnetic field in bulk regions (unit: A/m) or on interfaces (unit: A).

### Syntax of Command File and Parameter File

PMIs for generic bulk or interface  $\vec{H}_{\text{eff}}$  contributions are activated by adding the following line to the corresponding region or interface `Physics` sections of the command file:

```
LLG(HEff( "<name1>" [ "<name2>" ... ] ))
```

Contributions from all selected models are added during assembly.

Model parameters (if any) are taken from the appropriate region-specific or interface-specific sections of the `.par` file.

### Base Class for Generic Bulk or Interface for Effective Magnetic Field PMIs

PMI models for generic bulk or interface  $\vec{H}_{\text{eff}}$  contributions are derived from the base class `PMI_LLGHeff_Base`:

```
///! Base-class for local contributions to H_eff in the LLG equation
class PMI_LLGHeff_Base : public PMI_Device_Base {
public:
    class Input : public PMI_Device_Input_Base {
public:
    ///! Constructor
    Input(const PMI_Device_Base* );
    ///! Location types for PMI_LLGHeff_Base
    enum locT {
        UNDEFINED_LOCATION,      ///< Nothing (only used as initial value)
        DOMAIN_INTERFACE,        ///< Subset of a mesh interface (METIS
                               // domain)
        MESH_INTERFACE,          ///< Full mesh interface
        DOMAIN_BULK,             ///< Subset of a mesh bulk region (METIS
                               // domain)
        BULK                     ///< Full mesh bulk region
    };
}
```

## Chapter 39: Physical Model Interface

### Ferroelectrics and Ferromagnetics

```
locT locationType;      ///< Specifies how to interpret the
                        vertex list
int interfaceIndex;    ///< Mesh interface index (or -1 if
                        called for bulk)

//! Bulk region index; valid both during bulk and interface
assembly.
/**
 * For interface terms, this can be used to distinguish between
 * interior and exterior normal vectors: if \a regionIndex is
 * equal to region1 of the mesh interface, the vector returned
 * by ReadAveragedNormalVectorAtInterfaceVertex points away
 * from region \a regionIndex; otherwise it points into region
 * \a regionIndex.
 */
int regionIndex;

//! \a vertexList is only used for DOMAIN_INTERFACE and
DOMAIN_BULK
/**
 * \a locationType determines how bulk vertex indices for
 * assembly are obtained:
 *
 * DOMAIN_INTERFACE:
 *   Mesh()->regioninterface(interfaceIndex)
 *   ->vertex([vertexList[i]])->index()
 *   i = 0, ..., vertexList->size()-1
 * MESH_INTERFACE:
 *   Mesh()->regioninterface(interfaceIndex)->vertex(i)->index()
 *   i = 0, ..., Mesh()->regioninterface(locationIndex)->
 *       size_vertex()-1
 * DOMAIN_BULK:
 *   Mesh()->region(locationIndex)->vertex(vertexList[i])->
 *       index()
 *   i = 0, ..., vertexList->size()-1
 * BULK:
 *   Mesh()->region(locationIndex)->vertex(i)->index()
 *   i = 0, ..., Mesh()->region(locationIndex)->size_vertex()-1
 */
const std::vector<int> *vertexList;
}; //end of class PMI_LLGHeffBase::Input

class Output {
public:
    Output(const PMI_Device_Base *base);
    sdevice_pmi_float_vector Hx; // x-component of Heff at
                                // each vertex
    sdevice_pmi_float_vector Hy; // y-component of Heff at
                                // each vertex
    sdevice_pmi_float_vector Hz; // z-component of Heff at
                                // each vertex
};
```

## Chapter 39: Physical Model Interface

### Ferroelectrics and Ferromagnetics

```
PMI_LLGHeff_Base (const PMI_Device_Environment& env);
virtual ~PMI_LLGHeff_Base ();
virtual void compute (const Input& input, Output& output) = 0;

//! Get averaged normal vector for interface vertex
/**
 * @param[in] i - interface index
 * @param[in] v - interface vertex index on \a i
 * @returns pointer to the coordinates of the normal vector
 */
const double *
    ReadAveragedNormalVectorAtInterfaceVertex(int i, int v);
//! Read SaturationMagnetization for region ri
double ReadSaturationMagnetization(int ri);
};
```

The PMI implementer must provide the function:

```
compute(const Input& input, Output& output)
```

This function is called once per parallel (bulk or interface) domain of the device, and the `compute()` function is expected to provide values for the `Hx`, `Hy`, and `Hz` fields of the `output` object at each global vertex in the current domain.

### Example: Exchange Bias

Typical anisotropy models do not distinguish between  $\vec{m}$  and  $-\vec{m}$ . Interfaces between ferromagnetic and anti-ferromagnetic layers, however, might break the symmetry between parallel and anti-parallel alignment of the magnetization directions on either side of the interface.

This effect is known as *exchange bias*, which can be described by an interface contribution to  $\vec{H}_{\text{eff}}$  of the form  $I_{\text{bias}} \vec{d}_{\text{bias}}$ , where:

- $I_{\text{bias}}$  (`I_bias` in the `.par` file) describes the strength of the exchange bias (unit: A, corresponding to an interface density of magnetic dipoles as discussed above).
- $\vec{d}_{\text{bias}}$  (`biasDir` in the `.par` file) is the bias direction.

Positive values of  $I_{\text{bias}}$  correspond to the case that favors alignment of the magnetization at the surface of the ferromagnetic layer parallel to  $\vec{d}_{\text{bias}}$ .

### Implementation of the Exchange Bias PMI

To implement the exchange bias model, you must declare a derived class:

```
#include <PMI.h>
#include <cmath>
class PMI_ExchangeBias : public PMI_LLGHeff_Base {
public:
    PMI_ExchangeBias(const PMI_Device_Environment& env);
```

## Chapter 39: Physical Model Interface

### Ferroelectrics and Ferromagnetics

```
void computeForInterfaceVertex(int ii, int ivi,
                               PMI_LLGH_eff_Base::Output &out);
void compute(const PMI_LLGH_eff_Base::Input &in,
             PMI_LLGH_eff_Base::Output &out);
private:
    std::vector<double> biasDir; // < direction of the bias
                                   field (normalized)
    double Ibias; // < J/T / m^2 = A (surface density of
                   magnetic dipoles)
};
```

The constructor of the derived class reads the model parameters from the .par file:

```
PMI_ExchangeBias::PMI_ExchangeBias(const PMI_Device_Environment& env)
: PMI_LLGH_eff_Base(env), biasDir(3)
{
    Ibias = InitParameter("I_bias", 0.0);
    InitParameter("biasDir", biasDir);
    if (biasDir.size() != 3) {
        printf("PMI model ExchangeBias --- biasDir must be a
               3D vector!\n");
        exit(1);
    }
    // convert biasDir to unit vector
    double length = std::sqrt(biasDir[0] * biasDir[0] +
                               biasDir[1] * biasDir[1] +
                               biasDir[2] * biasDir[2]);
    if (length == 0) {
        printf("PMI model ExchangeBias --- "
               "biasDir must be a non-zero 3D vector!\n");
        exit(1);
    }
    biasDir[0] /= length;
    biasDir[1] /= length;
    biasDir[2] /= length;
}
```

During model evaluation, Sentaurus Device calls the `compute()` function. Exchange bias is an interface effect. Therefore, `in.locationType` must refer to an interface; other location types are rejected:

```
void PMI_ExchangeBias::compute(const PMI_LLGH_eff_Base::Input &in,
                               PMI_LLGH_eff_Base::Output &out)
{
    switch(in.locationType) {
        case PMI_LLGH_eff_Base::Input::DOMAIN_INTERFACE:
            if (!in.vertexList) {
                printf("PMI_ExchangeBias on DOMAIN_INTERFACE needs a
                       vertexList!\n");
                exit(1);
            }
            for (int i = 0; i < in.vertexList->size(); i++) {
```

## Chapter 39: Physical Model Interface

### Ferroelectrics and Ferromagnetics

```
        computeForInterfaceVertex(in.interfaceIndex, (*in.vertexList)[i],
                                  out);
    }
    break;
case PMI_LLGHeff_Base::Input::MESH_INTERFACE:
{
    size_t n = Mesh()->regioninterface(in.interfaceIndex)->
               size_vertex();
    for (size_t i = 0; i < n; i++) {
        computeForInterfaceVertex(in.interfaceIndex, i, out);
    }
}
break;
default:
printf("PMI_ExchangeBias does not support locationType=%d\n",
       in.locationType);
exit(1);
}
}
```

The calculation for each selected interface vertex is performed by the function

`computeForInterfaceVertex(ii, vi, &out)`, where `ii` denotes the index of the current mesh interface, and `vi` is the vertex index relative to this interface. The `out` object provides storage for the resulting effective magnetic field:

```
void PMI_ExchangeBias::computeForInterfaceVertex(int ii, int ivi,
                                                 PMI_LLGHeff_Base::Output &out)
{
    const des_mesh *mesh = Mesh();
    const des_regioninterface* interface = Mesh()->regioninterface(ii);
    size_t bulk_vi = interface->vertex(iv_i)->index();
    // Unit of surface effective magnetic field: A (not A/m as in bulk)
    out.Hx[bulk_vi] = Ibias * biasDir[0];
    out.Hy[bulk_vi] = Ibias * biasDir[1];
    out.Hz[bulk_vi] = Ibias * biasDir[2];
}
```

Finally, you must provide a so-called virtual constructor function, which allocates a variable of the new class:

```
extern "C"
PMI_LLGHeff_Base* new_PMI_LLGHeff_Base(const PMI_Device_Environment&
                                         env) {
    return new PMI_ExchangeBias(env);
}
```

## Chapter 39: Physical Model Interface

### Ferroelectrics and Ferromagnetics

#### Note:

This function must have C linkage and exactly the same name as declared in the `PMI.h` header file.

This model is included in the installation of Sentaurus Device under the PMI model name `ExchangeBias`, which corresponds to the C++ file name `ExchangeBias.C`.

#### Example: Interface Anisotropy

Frequently, an interface between a ferromagnetic material and adjacent materials gives rise to a contribution to the energy density of the magnetic system that favors perpendicular alignment of the magnetization over in-plane alignment. The interface anisotropy model describes this effect in terms of a surface energy density proportional to the square of the scalar product of the magnetization direction  $\vec{m}$  and the surface normal direction  $\vec{n}$ . The resulting effective magnetic field contribution takes the form  $-I_{\text{aniso}} \vec{n}(\vec{n} \cdot \vec{m})$  (unit: A, corresponding to an interface density of magnetic dipoles as discussed above). Positive values of  $I_{\text{aniso}}$  favor an out-of-plane magnetization direction; negative values favor in-plane magnetization.

The interface anisotropy model can describe the transition from in-plane magnetic alignment in magnetic thin films of moderate thickness to perpendicular magnetic alignment in very thin films. For positive  $I_{\text{aniso}}$  (`I_aniso` in the `.par` file, unit: A), there is competition between an effective bulk anisotropy term due to the geometry, which favors in-plane magnetic alignment, and the interface anisotropy, which favors out-of-plane alignment. With decreasing film thickness, the relative importance of the interface term grows. For very thin films, the interface term dominates, resulting in perpendicular magnetic alignment.

#### Implementation of the Interface Anisotropy PMI

Like the exchange bias model, the interface anisotropy model is derived from `PMI_LLGHeff_Base`:

```
#include <PMI.h>
#include <cmath>

class PMI_InterfaceAnisotropy : public PMI_LLGHeff_Base {
public:
    PMI_InterfaceAnisotropy(const PMI_Device_Environment& env);
    void computeForInterfaceVertex(int ii, int ivi,
                                   PMI_LLGHeff_Base::Output &out);
    void compute(const PMI_LLGHeff_Base::Input &in,
                PMI_LLGHeff_Base::Output &out);
private:
    double I_aniso;           /// $J/T/m^2 = A$  (surface density of
                             // magnetic dipoles)
    const pmi_float* mx;     /// $x$ -component of magnetization direction
```

## Chapter 39: Physical Model Interface

### Ferroelectrics and Ferromagnetics

```
    const pmi_float* my; ///< y-component of magnetization direction
    const pmi_float* mz; ///< z-component of magnetization direction
};
```

Again, the constructor reads the model parameter:

```
PMI_InterfaceAnisotropy::  
PMI_InterfaceAnisotropy(const PMI_Device_Environment& env)  
: PMI_LLGHeff_Base(env)  
{ I_aniso = InitParameter("I_aniso", 0.0); }
```

For the most part, the `compute()` function of the interface anisotropy model is identical to that of the exchange bias model. However, there is one important difference: The effective field of the interface anisotropy model depends on the magnetization density.

Therefore, the `compute()` function of `PMI_InterfaceAnisotropy` must read the magnetization data:

```
void PMI_InterfaceAnisotropy::compute(const PMI_LLGHeff_Base::Input  
                                      &in, PMI_LLGHeff_Base::Output &out)  
{  
    sdevice_data *data = Data();  
    mx = data->ReadScalar(sdevice_data::vertex, "MagnetizationDir_x");  
    my = data->ReadScalar(sdevice_data::vertex, "MagnetizationDir_y");  
    mz = data->ReadScalar(sdevice_data::vertex, "MagnetizationDir_z");  
    switch(in.locationType) {  
        ... the rest of the function as in PMI_ExchangeBias ...  
    }  
}
```

The `computeForInterfaceVertex()` function handles the calculation of the scalar product between the surface normal vector and the magnetization direction. Note how the components of the magnetization vector `m[]` are set from the `mx`, `my`, and `mz` components. This provides the correct (local) derivatives of the effective field contribution to the LLG equation:

```
void  
PMI_InterfaceAnisotropy::  
computeForInterfaceVertex(int ii, int ivi,  
                           PMI_LLGHeff_Base::Output &out)  
{  
    const des_mesh *mesh = Mesh();  
    const des_regioninterface* interface = Mesh()->regioninterface(ii);  
    size_t bulk_vi = interface->vertex(ivи)->index();  
    const double *n = ReadAveragedNormalVectorAtInterfaceVertex(ii,  
                                                               ivи);  
  
    pmi_float m[3];  
    m[0] = pmi_float(mx[bulk_vi].get_value<double>(), 3, 0);  
    m[1] = pmi_float(my[bulk_vi].get_value<double>(), 3, 1);  
    m[2] = pmi_float(mz[bulk_vi].get_value<double>(), 3, 2);  
  
    pmi_float dot_prod = n[0] * m[0] + n[1] * m[1] + n[2] * m[2];
```

## Chapter 39: Physical Model Interface

### Ferroelectrics and Ferromagnetics

```
// Unit of surface effective magnetic field: A (not A/m as in bulk)
out.Hx[bulk_vi] = -I_aniso * dot_prod * n[0];
out.Hy[bulk_vi] = -I_aniso * dot_prod * n[1];
out.Hz[bulk_vi] = -I_aniso * dot_prod * n[2];
}
```

The implementation is completed by the definition of the virtual constructor:

```
extern "C"
PMI_LLGHeff_Base* new_PMI_LLGHeff_Base(const
    PMI_Device_Environment& env) {
    return new PMI_InterfaceAnisotropy(env);
}
```

#### Note:

This model is included in the installation of Sentaurus Device under the PMI model name `InterfaceAnisotropy`.

### Example: Local Demagnetizing Field

This model implements a local expression for the demagnetizing field in terms of a diagonal demagnetizing tensor  $\underline{N} = \text{diag}(N_x, N_y, N_z)$ :  $\underline{M}_{\text{eff, demag}} = -\underline{N}\underline{M} = -\underline{M}_{\text{sat}}\underline{N}$ .

### Implementation of the Local Demagnetizing Field PMI

The model is derived from the base class `PMI_LLGHeff_Base`:

```
#include <PMI.h>
class LocalDemagnetizingField : public PMI_LLGHeff_Base {
public:
    LocalDemagnetizingField(const PMI_Device_Environment& env);
    void computeForBulkVertex(int ri, int vi, PMI_LLGHeff_Base::Output
        &out);
    void compute(const PMI_LLGHeff_Base::Input &in,
                PMI_LLGHeff_Base::Output &out);
private:
    const pmi_float* mx; ///  
 x-component of magnetization direction
    const pmi_float* my; ///  
 y-component of magnetization direction
    const pmi_float* mz; ///  
 z-component of magnetization direction
    double Nx; // demagnetizing factor along x-axis
    double Ny; // demagnetizing factor along y-axis
    double Nz; // demagnetizing factor along z-axis
};
```

As usual, the constructor of the derived class reads the model parameters from the `.par` file:

```
LocalDemagnetizingField::
LocalDemagnetizingField(const PMI_Device_Environment& env)
: PMI_LLGHeff_Base(env)
{
    Nx = InitParameter("Nx", 0.0);
```

## Chapter 39: Physical Model Interface

### Ferroelectrics and Ferromagnetics

```
Ny = InitParameter("Ny", 0.0);
Nz = InitParameter("Nz", 0.0);
}
```

In contrast to the exchange bias model and the interface anisotropy model, the local demagnetizing field model implements a bulk contribution to the effective magnetic field. Consequently, the `compute()` function now requires `in.locationType` to refer to a bulk region or domain:

```
void LocalDemagnetizingField::compute(const PMI_LLGH_eff_Base::Input
                                         &in, PMI_LLGH_eff_Base::Output &out)
{
    // Get magnetization direction
    sdevice_data *data = Data();
    mx = data->ReadScalar(sdevice_data::vertex, "MagnetizationDir_x");
    my = data->ReadScalar(sdevice_data::vertex, "MagnetizationDir_y");
    mz = data->ReadScalar(sdevice_data::vertex, "MagnetizationDir_z");

    switch(in.locationType) {
    case PMI_LLGH_eff_Base::Input::DOMAIN_BULK:
        if (!in.vertexList) {
            printf("LocalDemagnetizingField on DOMAIN_BULK needs a
                   vertexList!\n");
            exit(1);
        }
        for (size_t i = 0; i < (*in.vertexList).size(); i++) {
            computeForBulkVertex(in.regionIndex, (*in.vertexList)[i],
                                 out);
        }
        break;
    default:
        printf("LocalDemagnetizingField does not support
               locationType=%d\n", in.locationType);
        exit(1);
    }
}
```

The evaluation of the effective magnetic field contribution at each bulk node is handled by the `computeForBulkVertex(ri, vi, &out)` function. The argument `ri` is the region index, and the argument `vi` is the global vertex index of the evaluation point. The region index is needed to query the saturation magnetization:

```
void
LocalDemagnetizingField::computeForBulkVertex(int ri, int vi,
                                              PMI_LLGH_eff_Base::Output &out)
{
    pmi_float m[3];
    m[0] = pmi_float(mx[vi].get_value<double>(), 3, 0);
    m[1] = pmi_float(my[vi].get_value<double>(), 3, 1);
    m[2] = pmi_float(mz[vi].get_value<double>(), 3, 2);

    double Msat = ReadSaturationMagnetization(ri);
```

## Chapter 39: Physical Model Interface

### Ferroelectrics and Ferromagnetics

```
// Effective magnetic field A/m
out.Hx[vi] = -Nx * Msat * m[0];
out.Hy[vi] = -Ny * Msat * m[1];
out.Hz[vi] = -Nz * Msat * m[2];
}
```

Finally, the virtual constructor must be defined:

```
extern "C"
PMI_LLGHeff_Base* new_PMI_LLGHeff_Base(const PMI_Device_Environment&
env) {
    return new LocalDemagnetizingField(env);
}
```

#### Note:

This model is included in the installation of Sentaurus Device under the PMI model name `LocalDemagnetizingField`.

## User-Defined Magnetostatic Potential Calculation

For the special case of an effective magnetic field contribution that can be written as the gradient of a magnetostatic potential,  $\vec{H}_{\text{longitudinal}} = -\vec{\nabla}\varphi_{\text{max}}$ , you can reuse the base class `PMI_LLGHeff_Base` to calculate a magnetostatic potential instead of a magnetic field (see [Base Class for Generic Bulk or Interface for Effective Magnetic Field PMIs on page 1421](#)).

In this mode of operation, the `compute()` function does not populate the fields of the `output` object with magnetic field values. Instead, it prepares an array containing the magnetostatic potential for each vertex (unit:  $\mu\text{m} \cdot \text{A}/\text{m}$ ) and uses this to set the value of the `MagnetostaticPotential` field by calling the `sdevice_data::WriteScalar()` function.

The resulting magnetostatic potential and the corresponding magnetic field can be plotted like other fields in Sentaurus Device:

```
Plot {
    MagnetostaticPotential
    LongitudinalMagneticField/Element/Vector
}
```

## Syntax of Command File and Parameter File

This special mode is activated by adding the following statement to the global `Physics` section of the command file:

```
Magnetism(MagnetostaticPotentialPMI=<name>)
```

Parameters (if any) are read from the global section of the `.par` file, and the `locationType` in the `input` object is set to zero. Instead of being called in parallel for each parallel domain of the device, for the magnetostatic potential calculation, there is only one global call for the

## Chapter 39: Physical Model Interface

### Electrical Resistivity

entire device. User-defined parallelization, for example, using OpenMP, can be used to accelerate this call.

---

## Electrical Resistivity

This section presents the following PMIs:

- [Metal Resistivity](#)
- [Schottky Resistance](#)

---

## Metal Resistivity

The metal resistivity  $\rho = 1/\sigma$  can be defined by a PMI (see [Transport in Metals on page 295](#)).

The name of the PMI must be specified in the `Physics` section of the command file as follows:

```
Physics { MetalResistivity (pmi_model) }
```

The simplified interface `PMI_MetalResistivity_Base` supports the diffusion-reaction species interface (see [Reaction–Diffusion Species Interface \(Compute Scope\)](#) on page 1243).

## Dependencies

The metal resistivity depends on the variables:

t	Lattice temperature [K]
f	Absolute value of the electric field [ $V\text{cm}^{-1}$ ]

The PMI model must compute the following result:

Resist	Metal resistivity [ $\Omega\text{cm}$ ]
--------	---

In the case of the standard interface, the following derivatives must be computed as well:

dResistdt	Derivative of Resist with respect to t [ $\Omega\text{K}^{-1}\text{cm}$ ]
dResistdf	Derivative of Resist with respect to f [ $\Omega\text{V}^{-1}\text{cm}^2$ ]

## Chapter 39: Physical Model Interface

### Electrical Resistivity

## Standard C++ Interface

The following base class is declared in the file `PMIModels.h`:

```
class MetalResistivity : public PMI_MetalResistivity {

public:
    MetalResistivity (const PMI_Environment& env);
    virtual ~MetalResistivity () ;

    virtual void Compute_Resist
        (const double t,          // lattice temperature
         const double f,          // absolute value of electric field
         double& Resist);       // metal resistivity

    virtual void Compute_dResistdt
        (const double t,          // lattice temperature
         const double f,          // absolute value of electric field
         double& dResistdt);    // derivative of metal resistivity
                                // with respect to lattice temperature

    virtual void Compute_dResistdf
        (const double t,          // lattice temperature
         const double f,          // absolute value of electric field
         double& dResistdf);    // derivative of metal resistivity
                                // with respect to electric field
};

}
```

The following virtual constructor must be implemented:

```
typedef PMI_MetalResistivity* new_PMI_MetalResistivity_func
    (const PMI_Environment& env);
extern "C" new_PMI_MetalResistivity_func new_PMI_MetalResistivity;
```

## Simplified C++ Interface

The following base class is declared in the file `PMI.h`:

```
class PMI_MetalResistivity_Base : public PMI_Vertex_Base {

public:
    class Input : public PMI_Vertex_Input_Base {
public:
    pmi_float t;          // lattice temperature
    pmi_float f;          // absolute value of the electric field
};

    class Output {
public:
    pmi_float Resist;    // metal resistivity
};

    PMI_MetalResistivity_Base (const PMI_Environment& env);
}
```

## Chapter 39: Physical Model Interface

### Electrical Resistivity

```
virtual ~PMI_MetalResistivity_Base () ;

virtual void compute (const Input& input, Output& output) = 0 ;
};
```

The following virtual constructor must be implemented:

```
extern "C"
PMI_MetalResistivity_Base* new_PMI_MetalResistivity_Base
(const PMI_Environment& env);
```

## Example: Linear Metal Resistivity

The following C++ code implements linear metal resistivity:

```
#include "PMIModels.h"

class MetalResistivity : public PMI_MetalResistivity
{
    private:
        double R0, AlphaT;

    public:
        MetalResistivity (const PMI_Environment& env);
        ~MetalResistivity ();

        void Compute_Resist
            (const double t,          // lattice temperature
             const double f,          // absolute value of electric field
             double& Resist);        // metal resistivity

        void Compute_dResistdt
            (const double t,          // lattice temperature
             const double f,          // absolute value of electric field
             double& dResistdt);     // derivative of metal resistivity
                                     // with respect to lattice temperature

        void Compute_dResistdf
            (const double t,          // lattice temperature
             const double f,          // absolute value of electric field
             double& dResistdf);     // derivative of metal resistivity
                                     // with respect to electric field
    };

MetalResistivity::
MetalResistivity (const PMI_Environment& env) :
    PMI_MetalResistivity (env)
{ // Gold values
    R0      = InitParameter ("R0",      2.0400e-06); // [ohm*cm]
    AlphaT = InitParameter ("AlphaT", 4.0000e-03); // [1/K]
}

MetalResistivity::
```

## Chapter 39: Physical Model Interface

### Electrical Resistivity

```
~MetalResistivity ()
{ }

void MetalResistivity:::
Compute_Resist (const double t, const double f, double& Resist)
{
    Resist = R0*( 1 + AlphaT*( t - 273 ) );
}

void MetalResistivity:::
Compute_dResistdt (const double t, const double f, double& dResistdt)
{
    dResistdt = R0*AlphaT;
}

void MetalResistivity:::
Compute_dResistdf (const double t, const double f, double& dResistdf)
{
    dResistdf = 0;
}

extern "C"
PMI_MetalResistivity* new_PMI_MetalResistivity
    (const PMI_Environment& env)
{
    return new MetalResistivity(env);
}
```

---

## Schottky Resistance

The Schottky resistance model (see [Resistive Contacts on page 270](#) and [Resistive Interfaces on page 277](#)) emulates the behavior of a Schottky contact or interface. The Schottky resistance PMI allows users to define the contact- or interface-distributed Schottky resistance as an arbitrary function of lattice temperature, electron temperature, hole temperature, electron affinity, band gap, bandgap narrowing, conduction-band effective density-of-states, valence-band effective density-of-states, and effective intrinsic density.

The name of the PMI can be specified interface-wise or electrode-wise in the `Physics` section of the command file as follows:

```
Physics ( Electrode = "top2" ) {
    DistResist=SchottkyResist(pmi_schottkyresist1)
}

Physics(RegionInterface="r1/r5") {
    DistResist=SchottkyResist(pmi_schottkyresist2)
}
```

## Chapter 39: Physical Model Interface

### Electrical Resistivity

## Dependencies

The Schottky resistance  $R_d$  can depend on the following variables:

$$R_d = R_d(T, T_n, T_p, \chi, E_g, E_{\text{bgn}}, N_C, N_V, n_{i,\text{eff}}) \quad (1362)$$

where:

- $T$  is the lattice temperature [K].
- $T_n, T_p$  are the carrier temperatures [K].
- $\chi$  is the electron affinity [eV].
- $E_g$  is the band gap [eV].
- $E_{\text{bgn}}$  is the bandgap narrowing [eV].
- $N_C, N_V$  are the conduction band and the valence band density-of-states [ $\text{cm}^{-3}$ ].
- $n_{i,\text{eff}}$  is the effective intrinsic density [ $\text{cm}^{-3}$ ].

The PMI model must compute the following result:

$$R_d \quad [\Omega \text{cm}^2]$$

In the case of the standard interface, the following derivatives must be computed as well:

$$\text{d}R_d/\text{dT} \quad \text{Derivative of } R_d \text{ with respect to } T [\Omega \text{cm}^2 \text{K}^{-1}]$$

$$\text{d}R_d/\text{dT}_n \quad \text{Derivative of } R_d \text{ with respect to } T_n [\Omega \text{cm}^2 \text{K}^{-1}]$$

$$\text{d}R_d/\text{dT}_p \quad \text{Derivative of } R_d \text{ with respect to } T_p [\Omega \text{cm}^2 \text{K}^{-1}]$$

$$\text{d}R_d/\text{d}\chi \quad \text{Derivative of } R_d \text{ with respect to } \chi [\Omega \text{cm}^2 \text{eV}^{-1}]$$

$$\text{d}R_d/\text{d}E_g \quad \text{Derivative of } R_d \text{ with respect to } E_g [\Omega \text{cm}^2 \text{eV}^{-1}]$$

$$\text{d}R_d/\text{d}E_{\text{bgn}} \quad \text{Derivative of } R_d \text{ with respect to } E_{\text{bgn}} [\Omega \text{cm}^2 \text{eV}^{-1}]$$

$$\text{d}R_d/\text{d}N_C \quad \text{Derivative of } R_d \text{ with respect to } N_C [\Omega \text{cm}^{-1} \text{eV}^{-1}]$$

## Chapter 39: Physical Model Interface

### Electrical Resistivity

`dRd_dNv` Derivative of  $R_d$  with respect to  $N_V$  [ $\Omega\text{cm}^{-1}\text{eV}^{-1}$ ]

`dRd_dnieff` Derivative of  $R_d$  with respect to  $n_{i,\text{eff}}$  [ $\Omega\text{cm}^{-1}\text{eV}^{-1}$ ]

## Standard C++ Interface

The following base class is declared in the file `PMIModels.h`:

```
class PMI_SchottkyResistanc : public PMI_Vertex_Interface {
public:
    class Input {
    public:
        double t; // lattice temperature
        double tn; // electron temperature
        double tp; // hole temperature
        double affin; // electron affinity
        double Eg; // bandgap
        double Ebgm; // bandgap narrowing
        double nc; // conduction-band effective density of states
        double nv; // valence-band effective density of states
        double nie; // effective intrinsic density
    };

    class Output {
    public:
        double resist; // Schottky resistance
        double dresistdt; // temperature derivative
        double dresistdttn; // electron temperature derivative
        double dresistdtpp; // hole temperature derivative
        double dresistdaaffin; // electron affinity derivative
        double dresistdEg; // bandgap derivative
        double dresistdEbgm; // bandgap narrowing derivative
        double dresistdncc; // conduction-band effective state dens
                           // derivative
        double dresistdnv; // valence-band effective state dens
                           // derivative
        double dresistdnie; // intrinsic carrier density derivative
    };

    PMI_SchottkyResistance (const PMI_Environment& env);
    virtual ~PMI_SchottkyResistance () ;

    virtual void compute(const Input& input, Output& output) = 0;
};
```

## Chapter 39: Physical Model Interface

### Electrical Resistivity

The following virtual constructor must be implemented:

```
virtual extern "C"  
PMI_SchottkyResistance* new_PMI_SchottkyResistance(const  
PMI_Environment& env)
```

## Simplified C++ Interface

The following base class is declared in the file `PMI.h`:

```
class PMI_SchottkyResistance_Base : public PMI_Vertex_Base {  
public:  
    class Input : public PMI_Vertex_Input_Base {  
public:  
    Input (const PMI_SchottkyResistance_Base* schottkyresist_base,  
           const int vertex);  
    pmi_float t; // lattice temperature  
    pmi_float tn; // electron temperature  
    pmi_float tp; // hole temperature  
    pmi_float affin; // electron affinity  
    pmi_float Eg; // bandgap  
    pmi_float Ebgn; // bandgap narrowing  
    pmi_float nc; // conduction-band effective density of states  
    pmi_float nv; // valence-band effective density of states  
    pmi_float nie; // effective intrinsic density  
};  
    class Output {  
public:  
    pmi_float resist; // Schottky resistance  
};  
  
PMI_SchottkyResistance_Base (const PMI_Environment& env);  
  
virtual ~PMI_SchottkyResistance_Base ();  
  
virtual void compute (const Input& input, Output& output) = 0;  
};
```

The following virtual constructor must be implemented:

```
extern "C"  
PMI_SchottkyResistance_Base* new_PMI_SchottkyResistance_Base(const  
PMI_Environment& env);
```

## Example: Built-in Schottky Resistance

The following C++ code reimplements the built-in Schottky resistance model (simplified C++ interface):

```
#include <math.h>  
#include "PMI.h"  
  
class Builtin_SchottkyResistance: public PMI_SchottkyResistance_Base {
```

## Chapter 39: Physical Model Interface

### Electrical Resistivity

```
private:
    pmi_float ComputeSchottkyResistance(const Input& input);

public:
    Builtin_SchottkyResistance(const PMI_Environment& env);
    ~Builtin_SchottkyResistance();
    void compute(const Input& input, Output& output);
};

Builtin_SchottkyResistance::
Builtin_SchottkyResistance(const PMI_Environment& env) :
    PMI_SchottkyResistance_Base (env) {}

Builtin_SchottkyResistance::
~Builtin_SchottkyResistance() {}

pmi_float Builtin_SchottkyResistance::
ComputeSchottkyResistance(const Input& input) {
    // Planck's constant divided by 2pi in J*s
    const double h_bar = 1.05458866419688266838371913965e-34;
    // Epsilon 0 in As/Vcm
    const double eps0 = 8.8542e-14;
    // electron charge in C
    const double e0 = 1.602192e-19;
    // electron mass in kg
    const double m0 = 9.109534e-31;
    // Boltzmann constant in J/K
    const double kB = 1.380662e-23;

    pmi_float tempDEV = 300; // K
    pmi_float kT = kB*tempDEV/e0; // energy in eV

    pmi_float dop = input.ReadDoping(PMI_Donor) -
                    input.ReadDoping(PMI_Acceptor);
    pmi_float epsSEM = input.InitModelParameter("epsilon",
                                                "Epsilon", 1);
    pmi_float N = (dop > 0) ? dop : -dop;

    double rinf = 0;
    double PhiB = 0;
    double M = 0;

    if(dop > 0) { // q = -1
        rinf = input.InitModelParameter("Rinf_e", "SchottkyResistance",
                                        2.4000e-09);
        PhiB = input.InitModelParameter("PhiB_e", "SchottkyResistance",
                                        0.6);
        M = input.InitModelParameter("mt_e", "SchottkyResistance",
                                    0.19);
    } else { // q = 1
        rinf = input.InitModelParameter("Rinf_h", "SchottkyResistance",
                                        5.2000e-09);
```

## Chapter 39: Physical Model Interface

### Simulation Controls

```
PhiB = input.InitModelParameter("PhiB_h", "SchottkyResistance",
                                0.51);
M = input.InitModelParameter("mt_h", "SchottkyResistance",
                            0.16);
}

pmi_float Rinf = rinf*300/tempDEV;
pmi_float E00 = h_bar/2.0*100.0*sqrt(N/eps0/epsSEM/M/m0); // eV

pmi_float E0 = 0;
if(E00 < kT/100)
    E0 = kT;
else if(E00 > kT*100)
    E0 = E00;
else
    E0 = E00*cosh(E00/kT)/sinh(E00/kT);

return Rinf*exp(PhiB/E0);
}

void Builtin_SchottkyResistance::
compute(const Input& input, Output& output) {
    output.resist = ComputeSchottkyResistance(input);
}

extern "C"
PMI_SchottkyResistance_Base* new_PMI_SchottkyResistance_Base(
    const PMI_Environment& env) {
    return new Builtin_SchottkyResistance(env);
}
```

---

## Simulation Controls

This section presents the following PMIs:

- [Current Plot File](#)
- [Postprocessing for Transient Simulations](#)
- [Preprocessing for Newton Iterations and Newton Step Control](#)

---

## Current Plot File

The current plot PMI allows user-computed entries to be added to the current plot file. It is specified in the `CurrentPlot` section of the command file. For example:

```
CurrentPlot { pmi_CurrentPlot }
```

## Chapter 39: Physical Model Interface

### Simulation Controls

The interface has access to the device mesh and device data (see [Runtime Support for Mesh-Based PMI Models on page 1245](#)).

See [Current Plot File on page 1491](#) for a Tcl-based alternative to the current plot PMI.

## Structure of Current Plot File

A current plot file consists of a header section and a data section. For each function, the structure can be described as follows:

```
dataset name
function name
value0
value1
...
```

A dataset name denotes a dataset. For example:

```
time
Tmin
```

If a dataset corresponds to a region or contact, then it is customary to add the region or contact name:

```
gate Charge
```

The function name describes the function. For example:

```
ElectrostaticPotential
Temperature
```

Afterwards, a function value is added to the current plot file for each plot time point.

## Standard C++ Interface

The following base class is declared in the file `PMIModels.h`:

```
class PMI_CurrentPlot : public PMI_Device_Interface {

public:
    PMI_CurrentPlot (const PMI_Device_Environment& env);
    virtual ~PMI_CurrentPlot () ;

    virtual void Compute_Dataset_Names
    (des_string_vector& dataset) = 0;

    virtual void Compute_Function_Names
    (des_string_vector& function) = 0;

    virtual void Compute_Plot_Values
    (des_double_vector& value) = 0;
};
```

## Chapter 39: Physical Model Interface

### Simulation Controls

The `Compute_Dataset_Names()` and `Compute_Function_Names()` methods generate the header in the current plot file (see [Structure of Current Plot File on page 1440](#)). The method `Compute_Plot_Values()` is called for each plot time point to compute the plot values. Use the `push_back()` function to add values to the arrays dataset, function, or value.

#### Note:

These three methods must always compute the same number of values. Otherwise, an inconsistent current plot file will be generated.

The prototype for the virtual constructor is given as:

```
typedef PMI_CurrentPlot* new_PMI_CurrentPlot_func  
    (const PMI_Device_Environment& env);  
extern "C" new_PMI_CurrentPlot_func new_PMI_CurrentPlot;
```

## Simplified C++ Interface

The following base class is declared in the file `PMI.h`:

```
class PMI_CurrentPlot_Base : public PMI_Device_Base {  
  
public:  
    class Input : public PMI_Device_Input_Base {  
public:  
    };  
  
    class Output_Header {  
public:  
        des_string_vector dataset; // array of dataset names  
        des_string_vector function; // array of function names  
    };  
  
    class Output_Body {  
public:  
        sdevice_pmi_float_vector value; // array of plot values  
    };  
  
    PMI_CurrentPlot_Base (const PMI_Device_Environment& env);  
    virtual ~PMI_CurrentPlot_Base ();  
  
    virtual void compute_header (Output_Header& output) = 0;  
    virtual void compute_body (const Input& input, Output_Body&  
                           output) = 0;  
};
```

The prototype for the virtual constructor is given as:

```
typedef PMI_CurrentPlot_Base* new_PMI_CurrentPlot_Base_func  
    (const PMI_Device_Environment& env);  
extern "C" new_PMI_CurrentPlot_Base_func new_PMI_CurrentPlot_Base;
```

## Example: Average Electrostatic Potential

The following example computes regionwise averages for the electrostatic potential. This is the same functionality as provided by the built-in current plot command (see [Tracking Additional Data in the Current File on page 165](#)):

```
class CurrentPlot : public PMI_CurrentPlot {
private:
    typedef std::vector<des_bulk*> des_bulk_vector;

    const des_mesh* mesh;           // device mesh
    des_bulk_vector regions;       // list of semiconductor bulk regions
    double scale;                 // scaling factor

public:
    CurrentPlot (const PMI_Device_Environment& env);
    ~CurrentPlot ();

    void Compute_Dataset_Names (des_string_vector& dataset);
    void Compute_Function_Names (des_string_vector& function);
    void Compute_Plot_Values (des_double_vector& value);
};

CurrentPlot::
CurrentPlot (const PMI_Device_Environment& env) :
    PMI_CurrentPlot (env)
{ mesh = Mesh ();
  // determine regions to process
  for (size_t ri = 0; ri < mesh->size_region (); ri++) {
    des_region* r = mesh->region (ri);
    if (r->type () == des_region::bulk) {
      des_bulk* b = dynamic_cast <des_bulk*> (r);
      if (b->material () != "Oxide") {
        // we found a semiconductor bulk region
        regions.push_back (b);
      }
    }
  }
  // read parameters
  scale = InitParameter ("scale", 0.0);
}

CurrentPlot::
~CurrentPlot ()
{

void CurrentPlot::
Compute_Dataset_Names (des_string_vector& dataset)
{ for (size_t ri = 0; ri < regions.size (); ri++) {
    des_bulk* b = regions [ri];
    std::string name = "Average_";
    
```

## Chapter 39: Physical Model Interface

### Simulation Controls

```
        name += b->name ();
        name += "ElectrostaticPotential";
        dataset.push_back (name);
    }
}

void CurrentPlot::
Compute_Function_Names (des_string_vector& function)
{ for (size_t ri = 0; ri < regions.size (); ri++) {
    function.push_back ("ElectrostaticPotential");
}
}

void CurrentPlot::
Compute_Plot_Values (des_double_vector& value)
{ des_data* data = Data ();
  const double*const* measure = data->ReadMeasure ();
  const double* pot = data->ReadScalar (des_data::vertex,
                                         "ElectrostaticPotential");
  for (size_t ri = 0; ri < regions.size (); ri++) {
    des_bulk* b = regions [ri];

    double sum_pot = 0.0;
    double sum_measure = 0.0;

    for (size_t ei = 0; ei < b->size_element (); ei++) {
      des_element* e = b->element (ei);
      for (size_t vi = 0; vi < e->size_vertex (); vi++) {
        des_vertex* v = e->vertex (vi);
        const double m = measure [e->index ()][vi];
        const double p = pot [v->index ()];
        sum_pot += m * p;
        sum_measure += m;
      }
    }
    value.push_back (scale * (sum_pot / sum_measure));
  }
}

extern "C" {
PMI_CurrentPlot* new_PMI_CurrentPlot (const PMI_Device_Environment&
                                       env)
{ return new CurrentPlot (env);
}
}
```

## Postprocessing for Transient Simulations

The postprocess PMI allows you to post-compute data during a transient simulation. The PMI is called after a transient time step has succeeded. It is specified in the Math section of the command file.

For example:

```
Math {
    PostProcess ( Transient = "pmi_postprocess" )
}
```

The interface provides access to the device mesh and device data (see [Runtime Support for Mesh-Based PMI Models on page 1245](#)).

## Standard C++ Interface

The following base class is declared in the file `PMIModels.h`:

```
class PMI_PostProcess : public PMI_Device_Interface {

public:
    PMI_PostProcess (const PMI_Device_Environment& env);
    virtual ~PMI_PostProcess ();

    virtual void Compute_PostProcess () = 0;
}
```

The method `Compute_PostProcess()` is called after the transient time step has successfully completed.

## Simplified C++ Interface

The following base class is declared in the file `PMI.h`:

```
class PMI_PostProcess_Base : public PMI_Device_Base {

public:
    class Input : public PMI_Device_Input_Base {
    public:
    };

    class Output {
    public:
    };

    PMI_PostProcess_Base (const PMI_Device_Environment& env);
    virtual ~PMI_PostProcess_Base ();
```

## Chapter 39: Physical Model Interface

### Simulation Controls

```
    virtual void compute (const Input& input, Output& output) = 0;  
};
```

The prototype for the virtual constructor is given as:

```
typedef PMI_PostProcess_Base* new_PMI_PostProcess_Base_func  
    (const PMI_Device_Environment& env);  
extern "C" new_PMI_PostProcess_Base_func new_PMI_PostProcess_Base;
```

The method `compute()` is called after the transient time step has successfully completed.

## Example: Postprocess User-Field

The following code modifies the user-field depending on the current temperature change and transient step size after the transient time step has succeeded:

```
#include "PMIModels.h"  
  
class PostProcess : public PMI_PostProcess {  
  
private:  
    const des_mesh* mesh;  
  
public:  
    PostProcess (const PMI_Device_Environment& env);  
    ~PostProcess ();  
    void Compute_PostProcess ();  
};  
  
PostProcess::  
PostProcess (const PMI_Device_Environment& env) :  
    PMI_PostProcess (env)  
{  
    mesh = Mesh();  
}  
  
PostProcess::  
~PostProcess ()  
{  
}  
  
void PostProcess::  
Compute_PostProcess ()  
{  
    des_data* data = Data();  
  
    const double* T = data->ReadScalar(des_data::vertex,  
        "LatticeTemperature");  
    const double* T0 = data->ReadScalar(des_data::vertex,  
        "PMIUserField0");  
  
    double* delta_T = new double [mesh->size_vertex ()];
```

## Chapter 39: Physical Model Interface

### Simulation Controls

```
for (int vi=0; vi<mesh->size_vertex (); vi++) {
    delta_T[vi] = (T[vi]-T0[vi])/ReadTransientStepSize();
}
data->WriteScalar(des_data::vertex, "PMIUserField0", T);
data->WriteScalar(des_data::vertex, "PMIUserField1", delta_T);
if (delta_T!=NULL) { delete [] delta_T; }
}
```

---

## Preprocessing for Newton Iterations and Newton Step Control

The Newton step PMI allows you to precompute data during Newton iterations (using a nonlinear solver). This PMI is called after each Newton iteration. The interface provides access to the device mesh and device data (see [Runtime Support for Mesh-Based PMI Models on page 1245](#)). This PMI is specified in the `Coupled` section (subsection `PMI_NewtonStep`) of the command file. For example:

```
Coupled( PMI_NewtonStep( pmiNewtonStep(<parameter_list>) ) )
{ Poisson Electron Hole }
```

Here, `pmiNewtonStep` is the name of the PMI model. Different `Coupled` sections can use different PMI models. For example:

```
Solve {
    # "pmi_Newton1" is name the first PMI function
    Coupled( PMI_NewtonStep(pmi_Newton1) )
    { Poisson electron hole }

    Transient (
        InitialTime = 0.0 Finaltime = 1.0
        InitialStep = 0.01 MaxStep = 0.1 MinStep = 1.0e-06
    ) {# "pmi_Newton2" is name the second PMI function
        Coupled( PMI_NewtonStep( PMI_NewtonStep(pmi_Newton2
            (parameter_list)) ) )
        { Poisson Electron Hole Temperature }
    }

    Quasistationary (
        InitialStep=0.1 MaxStep=0.5 MinStep=0.001
        Increment=1.3
        Goal {name="drain" Voltage=0.6}
    )
    {# same PMI function
        Coupled( PMI_NewtonStep(pmi_Newton2(parameter_list)... ) )
        { poisson electron hole }
    }
}
```

## Standard C++ Interface

The following base class is declared in the file `PMIModels.h`:

```
class PMI_NewtonStep : public PMI_Device_Interface
{
protected:
    PMI_Newton::Info *info;

public:
    PMI_NewtonStep (const PMI_Device_Environment& env,
                    PMI_Newton::Info *i=0);
    virtual ~PMI_NewtonStep ();
    PMI_Newton::Info* getNewtonInfo() {
        return info;
    }

    // methods to be implemented by user
    virtual PMI_Result Compute (char* err_msg) = 0;
};
```

The method `Compute()` is called after each Newton iteration. The enumerator `PMI_Result` is defined in the file `PMIModels.h`:

```
enum PMI_Result {
    PMI_NextStep = 0,
    PMI_Converged = 1,
    PMI_Diverged = 2,
    PMI_Undefined
};
```

### Function `PMI_Newton::GetLogFile()` and Class `PMI_Newton::Info`

The `PMIModels.h` file contains the function `FILE* PMI_Newton::GetLogFile()`. It returns a pointer to the log file and, therefore, the PMI function can print messages to the log file. See the relevant files of the following example:

```
$STROOT/tcad/$STRELEASE/lib/sdevice/src/pmi_NewtonStep
```

The `PMI_Newton::Info` class is an interface class between Sentaurus Device and `PMI_NewtonStep`, and it contains information about the Newton process. The function `getNewtonInfo()` returns the corresponding pointer. The important functions of this class are as follows (see [Standard C++ Interface on page 1447](#)):

- `IsTrans()`, `IsQstat()`, and `IsContin()` return a value equal to `true` if the Coupled statement is a subsection of the corresponding section: Transient, Quasistationary, or Continuation.
- `InitialStep()`, `MaxStep()`, `MinStep()`, `Decrement()`, `Increment()`, `InitialTime()`, `FinalTime()`, and `MaxIters()` return the corresponding values as in the command file.

## Chapter 39: Physical Model Interface

### Simulation Controls

- `UsedIters()`, `RHS()`, `NewtonStep()`, `Error()`, `TimeFrom()`, and `TimeTo()` return the corresponding values as in the log file.
- `std::vector<std::string> pdeName` contains the PDE name as in the log file.
- `ChangeTimeTo(double t)` performs PMI time-step control (see [Flow of Computation on page 1449](#)).

## Simplified C++ Interface

The following base class is declared in the file `PMI.h`:

```
class PMI_NewtonStep_Base : public PMI_Device_Base
{
protected:
    PMI_Newton::Info *info;

public:
    class Input : public PMI_Device_Input_Base {
    public:
        Input (const PMI_NewtonStep_Base* newtonstep_base);
    };

    class Output {
    public:
        char err_msg[256];
    };

    PMI_NewtonStep_Base(
        const PMI_Device_Environment& env, PMI_Newton::Info *i=0);
    virtual ~PMI_NewtonStep_Base ();

    PMI_Newton::Info* getNewtonInfo() {
        return info;
    }

    virtual PMI_Result compute (const Input& input, Output& output)
        = 0;
};
```

The prototype for the virtual constructor is given as:

```
typedef PMI_NewtonStep_Base* new_PMI_NewtonStep_Base_func
    (const PMI_Device_Environment& env, PMI_Newton::Info *i);
extern "C" new_PMI_NewtonStep_Base_func new_PMI_NewtonStep_Base;
```

The method `compute()` is called after each Newton iteration.

## Flow of Computation

The following example shows the Quasistationary command (see previous example):

```
Starting solve of next problem:  
Quasistationary (  
    Initial step in t: 0.1, Minimum step in t: 0.001, Maximum step  
    in t: 0.5,  
    ...  
) { Coupled ( PMI_NewtonStep(pmi_Newton2(parameter_list)) )  
    { Poisson Electron Hole }  
}  
// call constructor PMI_NewtonStep  
PMI_NewtonStep *pmi_Newton = new PMI_NewtonStep (...)  
bool Is_pmiConverged = false, Is_pmiDiverged = false;  
  
Computing step from t=0.4 to t=0.5  
// for this case PMI_Newton::Info functions return the  
// following values:  
// TimeFrom()==0.4 and TimeTo()==0.5  
Iteration |Rhs| |step| error  
-----  
0          3.89e+08  
1          1.21e+07   1.25e-03   3.12e+02  
...  
m          6.62e+06   3.36e-05   5.85e+00  
int pmiResult = pmi_Newton->Compute();  
if( pmiResult != pmiNextStep){  
    if(pmiResult == PMI_Converged) Is_pmiConverged = true;  
    if(pmiResult == PMI_Diverged) Is_pmiDiverged = true;  
}  
if(Is_sdeviceDiverged || Is_pmiDiverged) {  
    // If pmiResult==PMI_Diverged, then user have to use  
    // function ChangeTimeTo(new_t) (new_t <= TimeTo())  
    // In this case sdevice reduces the time step.  
    // It is PMI time step control  
    goto Iteration#0; // sdevice decrease step  
} else if(Is_sdeviceConverged && Is_pmiConverged)  
    goto Finish;  
} else {  
    goto NextNewtonStep  
}  
...  
Finished, because...  
Error smaller than 1 ( 7.3461E-02 ) and Is_pmiConverged == true  
// call destructor ~PMI_NewtonStep
```

---

## Various

This section presents the following PMIs:

- [Dielectric Permittivity](#)
  - [Energy Relaxation Times](#)
  - [Gamma Factor for Density Gradient Model](#)
  - [Heavy Ion Spatial Distribution](#)
  - [Hot-Carrier Injection](#)
  - [Incomplete Ionization](#)
  - [Space Factor](#)
- 

### Dielectric Permittivity

This PMI provides access to the dielectric permittivity  $\epsilon$  in [Equation 37 on page 229](#). To activate it, in the `Physics` section of the command file, specify:

```
Physics { DielectricConstant (<string>) }
```

where `<string>` is the name of the PMI model.

The PMI supports anisotropic dielectric permittivity, and the model can be evaluated along different crystallographic axes. The enumeration type `PMI_AnisotropyType` determines the axis. The default is isotropic dielectric permittivity.

If anisotropic dielectric permittivity is activated in the command file, then the PMI class `PMI_DielectricConstant` is also instantiated in the anisotropic direction.

### Dependencies

The dielectric permittivity can depend on the variable:

`t` Lattice temperature [K]

The PMI model must compute the following results:

`epsilon` Dielectric permittivity  $\epsilon$  relative to  $\epsilon_0$  [1]

## Chapter 39: Physical Model Interface

Various

In the case of the standard interface, the following derivative must be computed as well:

depsilondt    Derivative of  $\epsilon$  relative to  $t$  [1/K]

## Standard C++ Interface

The following base class is declared in the file `PMIModels.h`:

```
class PMI_DielectricConstant : public PMI_Vertex_Interface {  
public:  
    PMI_DielectricConstant (const PMI_Environment& env, const  
                           PMI_AnisotropyType anisotype);  
    virtual ~PMI_DielectricConstant ();  
    PMI_AnisotropyType AnisotropyType () const { return anisoType; }  
    virtual void Compute_epsilon (const double t, double& epsilon) = 0;  
    virtual void Compute_depsilondt (const double t, double&  
                                    depsilondt) = 0;  
};
```

The following virtual constructor must be implemented:

```
typedef PMI_DielectricConstant* new_PMI_DielectricConstant_func (const  
                     PMI_Environment& env, const PMI_AnisotropyType anisotype);  
extern "C" new_PMI_DielectricConstant_func new_PMI_DielectricConstant;
```

## Simplified C++ Interface

The following base class is declared in the file `PMI.h`:

```
class PMI_DielectricConstant_Base : public PMI_Vertex_Base {  
public:  
    class Input : public PMI_Vertex_Input_Base {  
    public:  
        pmi_float t; // lattice temperature  
    };  
  
    class Output {  
    public:  
        pmi_float epsilon; // dielectric permittivity  
    };  
  
    PMI_DielectricConstant_Base (const PMI_Environment& env, const  
                               PMI_AnisotropyType anisotype);  
    virtual ~PMI_DielectricConstant_Base ();  
    PMI_AnisotropyType AnisotropyType () const;  
    virtual void compute (const Input& input, Output& output) = 0;  
};
```

## Chapter 39: Physical Model Interface

Various

The prototype for the virtual constructor is given as:

```
typedef PMI_DielectricConstant_Base*
new_PMI_DielectricConstant_Base_func
(const PMI_Environment& env, const PMI_AnisotropyType anisotype);
extern "C" new_PMI_DielectricConstant_Base_func
new_PMI_DielectricConstant_Base;
```

## Example: Temperature-Dependent Dielectric Permittivity

The following C++ code implements the temperature-dependent dielectric permittivity:

$$\epsilon(T) = \epsilon_0 + \epsilon_1 \cdot \tanh\left(\frac{T - T_c}{T_0}\right) \quad (1363)$$

```
#include "PMIModels.h"
class TempDep_Dielectric : public PMI_DielectricConstant {

private:
    double eps0, eps1, Tc, T0;

public:
    TempDep_Dielectric (const PMI_Environment& env, const
        PMI_AnisotropyType anisotype);
    ~TempDep_Dielectric ();
    void Compute_epsilon (const double t, double& epsilon);
    void Compute_depsilon_dt (const double t, double& depsilon_dt);
};

TempDep_Dielectric::TempDep_Dielectric (const PMI_Environment& env,
    const PMI_AnisotropyType anisotype) : PMI_DielectricConstant (env,
    anisotype)
{ // default values
    eps0 = InitParameter ("eps0", 11.7);
    eps1 = InitParameter ("eps1", 0.0);
    Tc = InitParameter ("Tc", 300.0);
    T0 = InitParameter ("T0", 300.0);
}

TempDep_Dielectric::~TempDep_Dielectric() {}

void TempDep_Dielectric::Compute_epsilon (const double t, double&
    epsilon)
{ epsilon = eps0+eps1*tanh((t-Tc)/T0); }

void TempDep_Dielectric::Compute_depsilon_dt (const double t, double&
    depsilon_dt)
{ const double th = tanh((t-Tc/T0));
    depsilon_dt = eps1/T0*(1.0-th*th);
}
```

```
extern "C"
PMI_DielectricConstant* new_PMI_DielectricConstant (const
    PMI_Environment&
    env, const PMI_AnisotropyType anisotype)
{ return new TempDep_Dielectric (env, anisotype);
}
```

---

## Energy Relaxation Times

The model for the energy relaxation times  $\tau$  in [Equation 86 on page 253](#) and [Equation 87 on page 253](#) can be specified in the `Physics` section of the command file. The four available possibilities are:

```
Physics {
    EnergyRelaxationTimes (
        formula
        constant
        irrational
        pmi_model_name
    )
}
```

These entries have the following meaning:

<code>formula</code>	Use the value of <code>formula</code> in the parameter file (default)
<code>constant</code>	Use constant energy relaxation times ( <code>formula=1</code> )
<code>irrational</code>	Use the ratio of two irrational polynomials ( <code>formula=2</code> )
<code>pmi_model_name</code>	Call a PMI model to compute the energy relaxation times

## Dependencies

The energy relaxation time  $\tau$  can depend on the variable:

`ct` Carrier temperature [K]

### Note:

The parameter `ct` represents the electron temperature during the calculation of  $\tau_n$  and the hole temperature during the calculation of  $\tau_p$ .

## Chapter 39: Physical Model Interface

Various

The PMI model must compute the following results:

tau Energy relaxation time  $\tau$  [s]

In the case of the standard interface, the following derivative must be computed as well:

dtaudct Derivative of  $\tau$  with respect to  $ct$  [ $sK^{-1}$ ]

## Standard C++ Interface

The following base class is declared in the file `PMIModels.h`:

```
class PMI_EnergyRelaxationTime : public PMI_Vertex_Interface {  
  
public:  
    PMI_EnergyRelaxationTime (const PMI_Environment& env);  
  
    virtual ~PMI_EnergyRelaxationTime ();  
  
    virtual void Compute_tau  
        (const double ct, double& tau) = 0;  
  
    virtual void Compute_dtaudct  
        (const double ct, double& dtaudct) = 0;  
};
```

The following two virtual constructors must be implemented for electron and hole energy relaxation times:

```
typedef PMI_EnergyRelaxationTime* new_PMI_EnergyRelaxationTime_func  
    (const PMI_Environment& env);  
extern "C" new_PMI_EnergyRelaxationTime_func  
new_PMI_e_EnergyRelaxationTime;  
extern "C" new_PMI_EnergyRelaxationTime_func  
new_PMI_h_EnergyRelaxationTime;
```

## Simplified C++ Interface

The following base class is declared in the file `PMI.h`:

```
class PMI_EnergyRelaxationTime_Base : public PMI_Vertex_Base {  
  
public:  
    class Input : public PMI_Vertex_Input_Base {  
    public:  
        pmi_float ct; // carrier temperature  
    };  
  
    class Output {
```

## Chapter 39: Physical Model Interface

Various

```
public:
    pmi_float tau; // energy relaxation time
};

PMI_EnergyRelaxationTime_Base (const PMI_Environment& env);
virtual ~PMI_EnergyRelaxationTime_Base ();

virtual void compute (const Input& input, Output& output) = 0;
};
```

The prototype for the virtual constructor is given as:

```
typedef PMI_EnergyRelaxationTime_Base*
new_PMI_EnergyRelaxationTime_Base_func
(const PMI_Environment& env);
extern "C" new_PMI_EnergyRelaxationTime_Base_func
new_PMI_e_EnergyRelaxationTime_Base;
extern "C" new_PMI_EnergyRelaxationTime_Base_func
new_PMI_h_EnergyRelaxationTime_Base;
```

## Example: Constant Energy Relaxation Times

The following C++ code implements constant energy relaxation times:

```
#include "PMIModels.h"

class Const_EnergyRelaxationTime : public PMI_EnergyRelaxationTime {

protected:
    double tau_const;

public:
    Const_EnergyRelaxationTime (const PMI_Environment& env);

    ~Const_EnergyRelaxationTime ();

    void Compute_tau
        (const double ct, double& tau);

    void Compute_dtaudct
        (const double ct, double& dtaudct);

};

Const_EnergyRelaxationTime::
Const_EnergyRelaxationTime (const PMI_Environment& env) :
    PMI_EnergyRelaxationTime (env)
{
}

Const_EnergyRelaxationTime::
~Const_EnergyRelaxationTime ()
{
```

## Chapter 39: Physical Model Interface

Various

```
}
```

```
void Const_EnergyRelaxationTime::  
Compute_tau (const double ct, double& tau)  
{ tau = tau_const;  
}
```

```
void Const_EnergyRelaxationTime::  
Compute_dtaudct (const double ct, double& dtaudct)  
{ dtaudct = 0.0;  
}
```

```
class Const_e_EnergyRelaxationTime : public  
Const_EnergyRelaxationTime {  
  
public:  
    Const_e_EnergyRelaxationTime (const PMI_Environment& env);  
  
    ~Const_e_EnergyRelaxationTime () {}  
  
};  
  
Const_e_EnergyRelaxationTime::  
Const_e_EnergyRelaxationTime (const PMI_Environment& env) :  
    Const_EnergyRelaxationTime (env)  
{ tau_const = InitParameter ("tau_const_e", 0.3e-12);  
}  
  
class Const_h_EnergyRelaxationTime : public  
Const_EnergyRelaxationTime {  
  
public:  
    Const_h_EnergyRelaxationTime (const PMI_Environment& env);  
  
    ~Const_h_EnergyRelaxationTime () {}  
  
};  
  
Const_h_EnergyRelaxationTime::  
Const_h_EnergyRelaxationTime (const PMI_Environment& env) :  
    Const_EnergyRelaxationTime (env)  
{ tau_const = InitParameter ("tau_const_h", 0.25e-12);  
}  
  
extern "C"  
PMI_EnergyRelaxationTime* new_PMI_e_EnergyRelaxationTime  
(const PMI_Environment& env)  
{ return new Const_e_EnergyRelaxationTime (env);  
}  
  
extern "C"  
PMI_EnergyRelaxationTime* new_PMI_h_EnergyRelaxationTime
```

```
(const PMI_Environment& env)
{ return new Const_h_EnergyRelaxationTime (env);
}
```

## Gamma Factor for Density Gradient Model

The density gradient model (see [Density Gradient Model on page 362](#)) contains the fit factor:

$$\gamma = \gamma_0 \cdot \gamma_{\text{pmi}} \quad (1364)$$

where  $\gamma_0$  is the solution-independent value, and  $\gamma_{\text{pmi}}$  is dependent on the solution [4].

The name of the PMI can be specified regionwise, materialwise, or globally in the `Physics` section of the command file as follows:

```
Physics ( Material = "Silicon" ) {
    eQuantumPotential( Gamma(name=pmi_eGamma -EffectiveMass)
    hQuantumPotential( Gamma(name=pmi_hGamma)
}
```

This model has the optional flag `EffectiveMass` (default) or `-EffectiveMass`. If the option `-EffectiveMass` is activated, then the DOS mass that is used as the prefactor of the density gradient equation is replaced with the free electron mass (only in the quantum potential model).

## Dependencies

The  $\gamma_{\text{pmi}}$  can depend on the following variables:

$$\gamma_{\text{pmi}} = \gamma_{\text{pmi}}(c, T_c, E_{\text{normal}}, h) \quad (1365)$$

where:

- $c = n, p$  is the carrier density [ $\text{cm}^{-3}$ ] for the `eQuantumPotential` model and the `hQuantumPotential` models.
- $T_c$  is the carrier temperature [K].
- $E_{\text{normal}}$  is the electric field perpendicular to the interface [ $\text{Vcm}^{-1}$ ].
- $h$  is the layer thickness [ $\mu\text{m}$ ].

The PMI model must compute the following result:

Gamma [unitless]

## Chapter 39: Physical Model Interface

Various

In the case of the standard interface, the following derivatives must be computed as well:

- dGamma\_dc      Derivative of Gamma with respect to  $n, p$  [ $\text{cm}^3$ ]
- dGamma\_dt      Derivative of Gamma with respect to  $T_c$  [ $\text{K}^{-1}$ ]
- dGamma\_df      Derivative of Gamma with respect to  $E_{\text{per}}$  [ $\text{cmV}^{-1}$ ]
- dGamma\_dh      Derivative of Gamma with respect to  $h$  [ $\text{cmV}^{-1}$ ]

## Standard C++ Interface

The following base class is declared in the file `PMIModels.h`:

```
class PMI_QDDGamma : public PMI_Vertex_Interface {

public:
    PMI_QDDGamma(const PMI_Environment& env);
    virtual ~PMI_QDDGamma();

    virtual void Compute_gamma(
        const double c,           // carrier density
        const double t,           // carrier temperature
        const double f,           // Enormal to interface
        const double h,           // layer thickness
        double& gamma) = 0;       // gamma

    virtual void Compute_dgamma_dc
        (const double c,           // carrier density
        const double t,           // carrier temperature
        const double f,           // Enormal to interface
        const double h,           // layer thickness
        double& dgammadc) = 0;   // derivative of gamma with respect
                                // to carrier density

    virtual void Compute_dgamma_dt
        (const double c,           // carrier density
        const double t,           // carrier temperature
        const double f,           // Enormal to interface
        const double h,           // layer thickness
        double& dgammaddt) = 0;  // derivative with respect to carrier
                                // temperature

    virtual void Compute_dgamma_df
        (const double c,           // carrier density
        const double t,           // carrier temperature
        const double f,           // Enormal to interface
        const double h,           // layer thickness
        double& dgammadf) = 0;   // derivative with respect to Enormal
```

## Chapter 39: Physical Model Interface

Various

```
// to interface

virtual void Compute_dgamma_dh
    (const double c,           // carrier density
     const double t,           // carrier temperature
     const double f,           // Enormal to interface
     const double h,           // layer thickness
     double& dgammadh) = 0;   // derivative with respect to layer
                               // thickness
};
```

The following virtual constructor must be implemented:

```
typedef PMI_QDDGamma* new_PMI_QDDGamma_func
    (const PMI_Environment& env);
extern "C" new_PMI_QDDGamma_func      new_PMI_QDDGamma;
```

## Simplified C++ Interface

The following base class is declared in the file `PMI.h`:

```
class PMI_MSC_QDDGamma_Base : public PMI_MSC_Vertex_Base {
public:
    class Input : public PMI_Vertex_Input_Base {
public:
    Input (const PMI_MSC_QDDGamma_Base* msc_qddgamma_base,
           const int vertex);

    pmi_float n;    // electron density
    pmi_float p;    // hole density
    pmi_float T;   // lattice temperature
    pmi_float eT;  // electron temperature
    pmi_float hT;  // hole temperature
    std::vector<pmi_float> s; // phase fraction
};

    class Output {
public:
    pmi_float val; // solution-dependent gamma value
};

    PMI_MSC_QDDGamma_Base (const PMI_Environment& env,
                           const std::string& msconfig_name,
                           const int model_index);

    virtual ~PMI_MSC_QDDGamma_Base ();

    virtual void compute (const Input& input, Output& output) = 0;
};
```

## Chapter 39: Physical Model Interface

Various

The following virtual constructor must be implemented:

```
extern "C"  
PMI_QDDGamma_Base* new_PMI_QDDGamma_Base (const PMI_Environment& env);
```

### Example: Solution-Dependent Gamma Factor

The following C++ code implements a solution-dependent Gamma factor (simplified C++ interface):

```
#include <math.h>  
#include "PMI.h"  
  
// gamma = gxy*gc*gt*gf*gh  
// where:  
// gxy = g0 + ax*x/x0 + ay*y/y0;  
// gc = exp(xc/(1+xc)); xc = Sqr(log(c/c0))  
// gt = exp(xt); xt = (t-300)/cT0  
// gf = 1 + af*xf*xf; xf = (f-f1)/f0  
// gh = 1 + ah*(xh - 1); xh = h/h0  
  
class eQDDGamma : public PMI_QDDGamma_Base{  
private:  
    // see above  
    double g0, x0, y0, ax, ay;  
    double c0, cT0;  
    double h0, ah;  
    double f0, f1, af;  
  
    int formula;  
    enum {is_c=1, is_t=2, is_f=4, is_h=8};  
  
public:  
    eQDDGamma (const PMI_Environment& env);  
    ~eQDDGamma ();  
    void compute (const Input& input, Output& output);  
};  
  
eQDDGamma::  
eQDDGamma (const PMI_Environment& env) :  
    PMI_QDDGamma_Base (env)  
{  
    g0 = InitParameter("g0", 1.); // [1]  
    c0 = InitParameter("c0", 1e10); // [cm-3]  
    cT0 = InitParameter("cT0", 300.); // [K]  
    x0 = InitParameter("x0", 1.); // [um]  
    ax = InitParameter("ax", 0.); // [1]  
    y0 = InitParameter("y0", 1.); // [um]  
    ay = InitParameter("ay", 0.); // [1]  
    h0 = InitParameter("h0", 1.); // [um]  
    ah = InitParameter("ah", 0.); // [1]  
    f0 = InitParameter("f0", 1e7); // [V/cm]  
    f1 = InitParameter("f1", 1e7); // [V/cm]
```

## Chapter 39: Physical Model Interface

Various

```
af = InitParameter("af", 0.);           // [1]

formula = InitParameter("formula", 0);   // [1]
if(formula < 0) formula = 0;
if(formula > 15) formula = 15;

}

eQDDGamma::~eQDDGamma () {}

void eQDDGamma::
compute (const Input& input, Output& output)
{
    const pmi_float& c = input.c; // carrier density
    const pmi_float& t = input.t; // carrier temperature
    const pmi_float& f = input.f; // Enormal to interface
    const pmi_float& h = input.h; // layer thickness

    double x, y, z;
    input.ReadCoordinate (x, y, z);
    pmi_float gxy = g0 + ax*x/x0 + ay*y/y0;

    pmi_float gc = 1.;
    pmi_float gt = 1.;
    pmi_float gf = 1.;
    pmi_float gh = 1.;

    if(formula & is_c) {
        const pmi_float n = (c < 1e-4 ? 1e-4 : c);
        const pmi_float xc = log(n/c0)*log(n/c0);
        gc = exp(xc/(1.+xc));
    }
    if(formula & is_t) {
        const pmi_float xt = (t-300)/cT0;
        gt = exp(xt);
    }
    if(formula & is_f) {
        const pmi_float xf = (f - f1)/f0;
        gf = 1. + af*xf*xf;
    }
    if(formula & is_h) {
        const pmi_float xh = h/h0;
        gh = 1. + ah*(xh - 1.);
    }

    output.gamma = gxy*gc*gt*gf*gh;
}

extern "C"
PMI_QDDGamma_Base* new_PMI_QDDGamma_Base
(const PMI_Environment& env)
{
    return new eQDDGamma(env);}
```

---

## Heavy Ion Spatial Distribution

The spatial distribution function  $R(w, l, E)$  can be defined by a PMI (see [Carrier Generation by Heavy Ions on page 775](#)).

The name of the PMI must be specified in the `Physics` section of the command file as follows:

```
Physics {
    HeavyIon(
        SpatialShape = PMI_shape_name
    )
}
```

or:

```
Physics {
    HeavyIon(
        SpatialShape = PMI_shape_name(Energy = value)    # [eV]
    )
}
```

## Dependencies

The spatial distribution function  $R(w, l, E)$  depends on the following variables:

- w      Radius defined as the perpendicular distance from the track [cm]
- l      Coordinate along the track [cm]
- E      Energy of heavy ion [eV]

The PMI model must compute the following result:

- R      The value of the spatial distribution  $R(w, l, E)$  [1]

## Standard C++ Interface

The following base class is declared in the file `PMIModels.h`:

```
class PMI_SpatialDistributionFunction: public PMI_Vertex_Interface {
    // * The spatial distribution function:
    // *
    // *   R(w,l,E)
    // *
    // * where
    // *   - l is the coordinate along the particle path [um];
    // *   - w is radial coordinate orthogonal to l [um];
```

## Chapter 39: Physical Model Interface

Various

```
// * - E is energy of heavy ion [eV];

public:
    PMI_SpatialDistributionFunction (const PMI_Environment& env,
                                    const char* IonType);

    virtual ~PMI_SpatialDistributionFunction () ;

// user-defined name of heavy ion (see
Using the Alpha Particle Model on page 774)
const char* GetHeavyIonType () const;

// methods to be implemented by user
    virtual void Compute_R (double& R, const double w, const double l
                           = -1., const double E = -1.) = 0;

};
```

The prototype for the virtual constructor is given as:

```
typedef PMI_SpatialDistributionFunction*
new_PMI_SpatialDistributionFunction_func
(const PMI_Environment& env, const char* HeavyIonName);
new_PMI_SpatialDistributionFunction_func
    new_PMI_SpatialDistributionFunction;
```

## Simplified C++ Interface

The following base class is declared in the file `PMI.h`:

```
class PMI_SpatialDistributionFunction_Base : public PMI_Vertex_Base {

public:
    class Input : public PMI_Vertex_Input_Base {
public:
    pmi_float w; // radius (perpendicular distance from track)
    pmi_float l; // coordinate along track
    pmi_float E; // energy of heavy ion
};

    class Output {
public:
    pmi_float R; // spatial distribution
};

    PMI_SpatialDistributionFunction_Base (const PMI_Environment& env,
                                         const char* name);
    virtual ~PMI_SpatialDistributionFunction_Base ();

    const char* GetHeavyIonType () const;

    virtual void compute (const Input& input, Output& output) = 0;
};
```

## Chapter 39: Physical Model Interface

Various

The prototype for the virtual constructor is given as:

```
typedef PMI_SpatialDistributionFunction_Base*
    new_PMI_SpatialDistributionFunction_Base_func
        (const PMI_Environment& env, const char* HeavyIonName);
extern "C" new_PMI_SpatialDistributionFunction_Base_func
    new_PMI_SpatialDistributionFunction_Base;
```

## Example: Gaussian Spatial Distribution Function

The built-in Gaussian spatial distribution function (see [Carrier Generation by Heavy Ions on page 775](#)) can also be implemented as a PMI model:

```
#include "PMIModels.h"

class Gaussian_SpatialDistributionFunction : public
PMI_SpatialDistributionFunction {

public:
    Gaussian_SpatialDistributionFunction (const PMI_Environment& env,
        const char* HeavyIonName);
    ~Gaussian_SpatialDistributionFunction () ;

    void Compute_R
        (double& R, const double w, const double l, const double E);
};

Gaussian_SpatialDistributionFunction::
Gaussian_SpatialDistributionFunction (const PMI_Environment& env,
const char* HeavyIonName) :
    PMI_SpatialDistributionFunction (env, HeavyIonName)
{ }

Gaussian_SpatialDistributionFunction::
~Gaussian_SpatialDistributionFunction ()
{ }

void Gaussian_SpatialDistributionFunction::
Compute_R(double& R, const double w, const double l, const double E)
{
    // the unit w,l is [cm]
    // the unit E is [eV] (in this implementation not used)
    // R(w,l,E) = exp( -(w/wt(l))^2 )

    double wt = 1.e-4; // scaling factor lum = 1.e-4cm
    double x = w/wt;

    R = exp(-x*x);
}

extern "C"
PMI_SpatialDistributionFunction* new_PMI_SpatialDistributionFunction
```

## Chapter 39: Physical Model Interface

Various

```
(const PMI_Environment& env, const char* HeavyIonName)
{
    return new Gaussian_SpatialDistributionFunction (env, HeavyIonName);
}
```

---

## Hot-Carrier Injection

Sentaurus Device provides a PMI for implementing hot-carrier injection models and computing the injection currents defined by the model. It is activated in the `Physics` interface section of the command file, `GateCurrent` subsection:

```
Physics(MaterialInterface="Silicon/Oxide") {
    GateCurrent( PMI_model(electron) )
}
```

The model can be used for both carrier types with either `PMI_model(electron)` or `PMI_model()`. The interface has access to the device mesh and device data (see [Runtime Support for Vertex-Based Multistate Configuration–Dependent Models on page 1244](#)).

## Dependencies

A hot-carrier PMI model has no explicit dependencies. However, it can depend on any field at runtime using the access to device data. The model must compute:

`gCurr` Vector of region/interface arrays with hot-carrier injection current densities; each region/interface array consists of the current density in each vertex of a region interface.

## Standard C++ Interface

The following base class is declared in the file `PMIModels.h`:

```
class PMI_HotCarrierInjection : public PMI_Device_Interface {

protected:
    const PMI_CarrierType cType;

public:
    PMI_HotCarrierInjection (const PMI_Device_Environment& env,
                           const PMI_CarrierType cType);
    virtual ~PMI_HotCarrierInjection () ;

    virtual void Compute_gCurr
        (const des_regioninterface_vector& regioninterfaces,
         // region interfaces associated with the model
         des_array_vector& gCurr) = 0; // gate injection current in each
                                       // vertex of specified region
                                       // interfaces
};


```

## Chapter 39: Physical Model Interface

Various

Two virtual constructors are required for electron and hole hot injection:

```
typedef PMI_HotCarrierInjection* new_PMI_HotCarrierInjection_func
(const PMI_Device_Environment& env, const PMI_CarrierType carType);
extern "C" new_PMI_HotCarrierInjection_func
new_PMI_e_HotCarrierInjection;
extern "C" new_PMI_HotCarrierInjection_func
new_PMI_h_HotCarrierInjection;
```

## Simplified C++ Interface

The following base class is declared in the file `PMI.h`:

```
class PMI_HotCarrierInjection_Base : public PMI_Device_Base {

public:
    class Input : public PMI_Device_Input_Base {
    public:
        des_regioninterface_vector regioninterfaces; // region interfaces
                                                // associated with
                                                // model name
    };

    class Output {
    public:
        sdevice_array_vector gCurr; // gate injection current in each
                                    // vertex of specified region
                                    // interfaces
    };

    PMI_HotCarrierInjection_Base (const PMI_Device_Environment& env);
    virtual ~PMI_HotCarrierInjection_Base ();

    virtual void compute (const Input& input, Output& output) = 0;
};
```

The prototype for the virtual constructor is given as:

```
typedef PMI_HotCarrierInjection_Base*
new_PMI_HotCarrierInjection_Base_func
(const PMI_Device_Environment& env);
extern "C" new_PMI_HotCarrierInjection_Base_func
new_PMI_e_HotCarrierInjection_Base;
extern "C" new_PMI_HotCarrierInjection_Base_func
new_PMI_h_HotCarrierInjection_Base;
```

## Example: Lucky Hot-Carrier Injection

The following example reimplements the built-in lucky injection model for electrons using the hot-carrier injection PMI:

```
#include <math.h>
#include "PMIModels.h"

class PMI_LuckyModel : public PMI_HotCarrierInjection {
private:
    const des_mesh* mesh; //device mesh
    des_data* data; //device data
    const double*const* measure;
    const double*const* surface_measure;
public:
    PMI_LuckyModel(const PMI_Device_Environment& env,
                   const PMI_CarrierType carType);
    ~PMI_LuckyModel();
    void Compute_gCurr(const des_regioninterface_vector&
                       regioninterfaces, des_array_vector& gCurr);
};

PMI_LuckyModel::PMI_LuckyModel(const PMI_Device_Environment& env,
                               const PMI_CarrierType carType) :
    PMI_HotCarrierInjection(env, carType)
{
    mesh = Mesh();
    data = Data();
    measure = data->ReadMeasure();
    surface_measure = data->ReadSurfaceMeasure();
}

PMI_LuckyModel::~PMI_LuckyModel()
{
}

void PMI_LuckyModel::Compute_gCurr(const des_regioninterface_vector& regioninterfaces,
                                   des_array_vector& gCurr)
{
    //compute current for each des_regioninterface associated with model
    for(int inter=0; inter < regioninterfaces.size(); inter++) {
        for(int k=0; k < regioninterfaces.at(inter)->size_vertex(); k++)
            gCurr[inter][k] = 0.0;

        des_regioninterface* ri = regioninterfaces.at(inter);
        des_bulk* b1 = ri->bulk1();
        des_bulk* b2 = ri->bulk2();

        //recognize regioninterface or regioninterface category
    }
}
```

## Chapter 39: Physical Model Interface

Various

```
if(((b1->material() == "Silicon") && (b2->material() == "Oxide"))
||((b1->material() == "Oxide") && (b2->material() ==
"Silicon"))) {

    des_bulk* semReg;
    des_bulk* insReg;
    if((b1->material() == "Silicon") && (b2->material() == "Oxide"))
    {
        semReg = b1;
        insReg = b2;
    }
    if((b2->material() == "Silicon") && (b1->material() == "Oxide"))
    {
        semReg = b2;
        insReg = b1;
    }

    //read region interface constants
    double eLambd = 8.9000e-07;//eLsem
    double eLambdR = 6.2000e-06;//eLsemR
    double eOxLambd = 3.2000e-07;//eLins
    double eBarrierHeight = 3.1;//eBar0
    double eAlfa = 2.6000e-04;//eBL12
    double eBeta = 3.0000e-05;//eBL23

    //read DataEntries used in computation
    const double* pot =
        data->ReadScalar(des_data::vertex, "ElectrostaticPotential");
    const double* OxField =
        data->ReadScalar(des_data::vertex, "InsulatorElectricField");
    const double* Epsilon =
        data->ReadScalar(des_data::element, "DielectricConstant");
    const double* eCurrent = NULL;
    const double* eField = NULL;
    if(cType == PMI_Electron) {
        eCurrent = data->ReadScalar(des_data::vertex,
                                      "eCurrentDensity");
        eField = data->ReadScalar(des_data::vertex, "eEparallel");
    }

    //integrate over the corresponding semiconductor region
    for(size_t vi = 0; vi < semReg->size_vertex(); vi++) {
        des_vertex* rv = semReg->vertex(vi);

        //find the nearest interface vertex
        //((distance to interface)
        des_vertex* NearestInterVertex;
        int NearestInterVertexRIind;
        double NearestDistance = 1.0e50;
        for(size_t k=0; k < ri->size_vertex(); k++) {
            des_vertex* interVert = ri->vertex(k);
            bool isSemiconductor = false;
            for(size_t i = 0; i < interVert->size_region(); i++) {
```

## Chapter 39: Physical Model Interface

Various

```
des_bulk* b = dynamic_cast<des_bulk*>(interVert->
    region(i));
if(b->material() == "Silicon") {
    isSemiconductor = true;
}
}
if(isSemiconductor) {
    double dist = 0.0;
    for(int kk=0; kk < mesh->dim(); kk++)
        dist += (interVert->coord()[kk] - rv->coord()[kk])*
            (interVert->coord()[kk] - rv->coord()[kk]);
    dist = sqrt(dist);
    if(dist < NearestDistance) {
        NearestDistance = dist;
        NearestInterVertex = interVert;
        NearestInterVertexRIind = k;
    }
}
}//nearest interface vertex

//find the nearest gate contact vertex
//(distance from NearestIntVertex to gate contact)
des_vertex* NearestContVertex;
double NearestContDistance = 1.0e50;
for(size_t k=0; k < insReg->size_vertex(); k++) {
    des_vertex* vins = insReg->vertex(k);
    if(vins->size_region() > 1) {
        for(size_t vvr = 0; vvr < vins->size_region(); vvr++) {
            des_region* rr = vins->region(vvr);
            if(rr->type() == des_region::contact) {
                //contact vertex
                double dist = 0.0;
                for(int kk=0; kk < mesh->dim(); kk++)
                    dist += (vins->coord()[kk] -
                        NearestInterVertex->coord()[kk])*(
                            vins->coord()[kk] -
                            NearestInterVertex->coord()[kk]);
                dist = sqrt(dist);
                if(dist < NearestContDistance) {
                    NearestContDistance = dist;
                    NearestContVertex = vins;
                }
            }
        }
    }
}
}//nearest gate contact vertex

//coordinates and distances are in um
//transform to cm
double OxThickn = NearestContDistance*1.0e-4;
double DistToSurf = NearestDistance*1.0e-4;

double xEps[3] = {0.0, 0.0, 0.0};
```

## Chapter 39: Physical Model Interface

Various

```
for(int k = 0; k < mesh->dim(); k++)
    xEps[k] = 0.5*(NearestInterVertex->coord()[k]
                    + NearestContVertex->coord()[k]);
des_element* xEl;
//find the oxide element
for(size_t ei = 0; ei < insReg->size_element(); ei++) {
    des_element* e = insReg->element(ei);
    for(size_t evi = 0; evi < e->size_vertex(); evi++) {
        double minCoord[3] = {1.0e50, 1.0e50, 1.0e50};
        double maxCoord[3] = {-1.0e50, -1.0e50, -1.0e50};
        des_vertex* ev = e->vertex(evi);
        for(int i = 0; i < mesh->dim(); i++) {
            if(ev->coord()[i] < minCoord[i])
                minCoord[i] = ev->coord()[i];
            if(ev->coord()[i] > maxCoord[i])
                maxCoord[i] = ev->coord()[i];
        }
        for(int i = 0; i < mesh->dim(); i++) {
            if( (xEps[i] >= minCoord[i]) && (xEps[i]
                >= maxCoord[i]) ) {
                xEl = e;
            }
        }
    }
}
double OxConst = Epsilon[xEl->index()];
double OxImage0 = 1.6e-19/16/3.1452/8.85e-14/OxConst;

//for electrons
if(cType == PMI_Electron) {
    double eCur = 0.0;
    double eOxField = pot[NearestContVertex->index()]
                    -pot[NearestInterVertex->index()];
    eOxField = eOxField > 0
        ? OxField[NearestInterVertex->index()]
        : -OxField[NearestInterVertex->index()];
    double barrier;
    if(pot[NearestInterVertex->index()]
        < pot[NearestContVertex->index()])
        barrier = eBarrierHeight
            - eAlfa*sqrt(fabs(eOxField))
            - eBeta*pow(fabs(eOxField),2.0/3.0);
    else
        barrier = eBarrierHeight
            + (pot[NearestInterVertex->index()] -
               pot[NearestContVertex->index()])
            - eAlfa*sqrt(fabs(eOxField))
            - eBeta*pow(fabs(eOxField),2.0/3.0);
    if(barrier < 0) barrier = 0.0;
    double eBarrierLoc = barrier;
    double p1 = 0.0;
    double eEnergy = eField[rv->index()]*eLambd;
    if(eEnergy > 1.0e-30) {
```

## Chapter 39: Physical Model Interface

Various

```
p1 = 0.25;
if (eBarrierLoc > 1.0e-30)
    p1 = 0.25*eEnergy/eBarrierLoc*exp(-eBarrierLoc/eEnergy);
}
double p2 = exp(-DistToSurf/eLambd);
double eDistFromSurf = 1.0e30;
if (eOxField > 1.0e-30) eDistFromSurf = sqrt(OxImage0/
    eOxField);
double p3 = exp(-eDistFromSurf/eOxLambd);

//find the node measure
double node_measure = 0.0;
for(size_t ei = 0; ei < rv->size_element(); ei++) {
    des_element* e = rv->element(ei);
    des_bulk* bulk = e->bulk();
    if(bulk->material() == "Silicon") {
        for(size_t evi = 0; evi < e->size_vertex(); evi++) {
            des_vertex* ev = e->vertex(evi);
            if(ev->index() == rv->index()) {
                node_measure += measure[e->index()][evi];
            }
        }
    } //semiconductor element
} //node measure

//convert measure in cm^dim
node_measure = node_measure*pow(1.0e-4,mesh->dim());

eCur = eCurrent[rv->index()]*p1*p2*p3*node_measure/eLambdR;
double risurface =
    surface_measure[ri->index()][NearestInterVertexRIind];
//convert to cm^(dim-1)
risurface = risurface*pow(1.0e-4,(mesh->dim()-1));
gCurr[inter][NearestInterVertexRIind] += eCur/risurface;
}

//for holes
if(cType == PMI_Hole) {
}
}//end integration on semiconductor region
}//end model implementation
}//end loop over "PMI_HotCarrierInjection" model regioninterfaces
}

extern "C" {
PMI_HotCarrierInjection* new_PMI_e_HotCarrierInjection
(const PMI_Device_Environment& env, const PMI_CarrierType carType)
{
    return new PMI_LuckyModel(env, carType);
}
}

extern "C" {
```

## Chapter 39: Physical Model Interface

Various

```
PMI_HotCarrierInjection* new_PMI_h_HotCarrierInjection
    (const PMI_Device_Environment& env, const PMI_CarrierType carType)
{
    return new PMI_LuckyModel(env, carType);
}
```

---

## Incomplete Ionization

The ionization factors  $G_D(T)$  and  $G_A(T)$  (see [Incomplete Ionization Model on page 340](#)) can be defined by a PMI.

The name of the PMI should be specified in the `Physics` section of the command file as follows:

```
Physics {
    IncompleteIonization( Model( PMI_model_name( "Species_name1"
                                                Species_name2 ..." ) ) )
}
```

In addition, it is possible to have a PMI for each species separately:

```
Physics {
    IncompleteIonization(
        Model(
            PMI_model_name1( "Species_name1" )
            PMI_model_name2( "Species_name2" )
        )
    )
}
```

The species PMI parameters should be defined in the parameter file (see [Parameter File of Sentaurus Device on page 1232](#)).

## Dependencies

The ionization factors  $G_D(T)$  and  $G_A(T)$  can depend on the variable:

$t$  Lattice temperature [K]

The PMI model must compute the following results:

$g$  Ionization factor  $G(T)$

## Chapter 39: Physical Model Interface

Various

In the case of the standard interface, the following derivative must be computed as well:

$dgdt$  Derivative of  $G(T)$  with respect to  $T$

## Standard C++ Interface

The following base class is declared in the file `PMIModels.h`:

```
enum PMI_SpeciesType {
    PMI_acceptor,
    PMI_donor
};

class PMI_DistributionFunction : public PMI_Vertex_Interface {

private:
    const PMI_SpeciesType speciesType;
    const char* speciesName;

public:
    PMI_DistributionFunction (const PMI_Environment& env,
                             const char* name
                             const PMI_SpeciesType type = PMI_acceptor);

    virtual ~PMI_DistributionFunction ();

    PMI_SpeciesType SpeciesType () const { return speciesType; }
    const char* SpeciesName () const { return speciesName; }

    // read parameter from Sentaurus Device parameter file
    // (override for PMI_Vertex_Interface::ReadParameter)
    const PMIBaseParam* ReadParameter (const char* name) const;

    // initialize parameter from Sentaurus Device parameter
    // file or from default value (override for
    // PMI_Vertex_Interface::InitParameter)
    double InitParameter (const char* name, double defaultvalue) const;

    virtual void Compute_g
        (const double T,           // lattice temperature
         double& g) = 0;          // g = G(T)

    virtual void Compute_dgdt
        (const double T,           // lattice temperature
         double& dgdt) = 0;        // dgdt = G'(T)

};

}
```

## Chapter 39: Physical Model Interface

Various

The prototype for the virtual constructor is given as:

```
typedef PMI_DistributionFunction* new_PMI_DistributionFunction_func
    (const PMI_Environment& env);
extern "C" new_PMI_DistributionFunction_func
    new_PMI_DistributionFunction;
```

## Simplified C++ Interface

The following base class is declared in the file `PMI.h`:

```
class PMI_DistributionFunction_Base : public PMI_Vertex_Base {

public:
    class Input : public PMI_Vertex_Input_Base {
    public:
        pmi_float t; // lattice temperature
    };

    class Output {
    public:
        pmi_float g; // ionization factor
    };

    PMI_DistributionFunction_Base (const PMI_Environment& env,
                                  const char* name,
                                  const PMI_SpeciesType type = PMI_acceptor);
    virtual ~PMI_DistributionFunction_Base ();

    PMI_SpeciesType SpeciesType () const;
    const char* SpeciesName () const;

    const PMIBaseParam* ReadParameter (const char* name) const;
    double InitParameter (const char* name, double defaultvalue) const;

    virtual void compute (const Input& input, Output& output) = 0;
};
```

The prototype for the virtual constructor is given as:

```
typedef PMI_DistributionFunction_Base*
    new_PMI_DistributionFunction_Base_func
    (const PMI_Environment& env, const char* name,
     const PMI_SpeciesType type);
extern "C" new_PMI_DistributionFunction_Base_func
    new_PMI_DistributionFunction_Base;
```

## Example: Matsuura Incomplete Ionization

The following C++ code implements the Matsuura model [1] for dopant Al in SiC material:

$$G_A(T) = 4 \exp\left(\frac{\Delta E_A - E_{ex}}{kT}\right) \cdot g_1 + \sum_{r=2}^{\infty} g_r \exp\left(\frac{\Delta E_r - \Delta E_A}{kT}\right) \quad (1366)$$

where  $g_1$  is the ground-state degeneracy factor,  $g_r$  is the  $(r-1)$ -th excited state degeneracy factor, and  $\Delta E_r$  is the difference in energy between the  $(r-1)$ -th excited state level and  $E_V$ .  $\Delta E_r$  is given by the hydrogenic dopant model [2]:

$$\Delta E_r = 13.6 \cdot \frac{m^*}{m_0 \cdot \epsilon_s^2} \cdot \frac{1}{r^2} \quad [\text{eV}] \quad (1367)$$

where  $m^*$  is the hole effective mass in SiC, and  $\epsilon_s$  is the dielectric constant of SiC.

The acceptor level is described as [2]:

$$\Delta E_A = \Delta E_1 + E_{CCC} \quad (1368)$$

where  $E_{CCC}$  is the energy induced due to central cell corrections.

The ensemble average  $E_{ex}$  of the ground and excited state levels of the acceptor is given by [3]:

$$E_{ex} = \frac{(\Delta E_A - \Delta E_r)g_r \exp\left(-\frac{\Delta E_A - \Delta E_r}{kT}\right)}{g_1 + \sum_{r=2}^{\infty} g_r \exp\left(-\frac{\Delta E_A - \Delta E_r}{kT}\right)} \quad (1369)$$

The Matsuura model can be implemented as follows:

```
class Matsuura_DistributionFunction : public PMI_DistributionFunction
{
protected:
    const double kB_300;           // Boltzmann constant * 300 [eV]
    int nb_item;                  // number of item in sum
    double *gr, *dEr;
    double Eex, dEA, Eccc;

public:
    Matsuura_DistributionFunction (const PMI_Environment& env,
                                   const char* name,
                                   const PMI_SpeciesType type =
                                   PMI_acceptor);

    ~Matsuura_DistributionFunction ();

    void Compute_g
        (const double T,           // lattice temperature
```

## Chapter 39: Physical Model Interface

Various

```
double& g); // g = G(T)

void Compute_dgdt
  (const double T, // lattice temperature
   double& dgdt); // dgdt = G'(T)

double Compute_Eex(double T); // compute Eex(T)
Equation 1369 on page 1475
double Compute_dEexdT(double T); // compute dEex/dT

};

Matsuura_DistributionFunction::
Matsuura_DistributionFunction (const PMI_Environment& env,
                               const char* name,
                               const PMI_SpeciesType type) :
  PMI_DistributionFunction (env, name, type),
  kB_300(1.380662e-23*300./1.602192e-19),
  // kB*T0/e0 = 0.02585199527 [eV]
  Eex(0.)
{
  nb_item = InitParameter ("NumberOfItem", 1);
  Eccc = InitParameter ("Eccc", 0);

  if(nb_item < 1) {
    printf("ERROR; PMI model Matsuura_DistributionFunction: parameter
           NumberOfItem < 1 \n");
    exit(1);
  }
  gr = new double[nb_item];
  dEr = new double[nb_item];

  char str_r[6], name_gr[6];

  int r;
  for(r=0; r<nb_item; ++r) {
    name_gr[0] = 'g'; name_gr[1] = '\0';
    sprintf(str_r, "%d\0", r+1);
    strcat(name_gr, str_r);
    const PMIBaseParam* par = ReadParameter(name_gr);
    if(!par) {
      printf("ERROR; PMI model Matsuura_DistributionFunction: cannot
             read parameter %s \n", name_gr);
      gr[r] = 2;
    } else
      gr[r] = *par;
  }

  const PMIBaseParam* par = ReadParameter("dEl");
  if(!par) {
    // dE[r] = 13.6 * m_eff/m0/eps/eps/r/r
    Compute_dEr(nb_item, dEr);
  }
}
```

```
    } else {
        char name_dEr[6];
        for(r=0; r<nb_item; ++r) {
            strcpy(name_dEr, "dE");
            sprintf(str_r, "%d\0", r+1);
            strcat(name_dEr, str_r);
            const PMIBaseParam* par = ReadParameter(name_dEr);
            if(!par) {
                printf("ERROR; PMI model Matsuura_DistributionFunction: cannot
                       read parameter %s \n", name_dEr);
                exit(1);
            }
            dEr[r] = *par;
        }
        dEA = dEr[0] + Eccc;
    }

Matsuura_DistributionFunction::
~Matsuura_DistributionFunction ()
{
    delete[] gr;
    delete[] dEr;
}

void Matsuura_DistributionFunction::Compute_g
    (const double T,           // lattice temperature
     double& g)               // g = G(T)
{
    const double kT = kB_300*T/300;

    Eex = Compute_Eex(T);
    g = gr[0];

    for(int r=1; r<nb_item; ++r) {
        double delta = dEA - dEr[r];
        g += gr[r]*exp( -delta/kT );
    }
    g *= 4.*exp( (dEA-Eex)/kT );
}

void Matsuura_DistributionFunction::Compute_dgdt
    (const double T,           // lattice temperature
     double& dgdt)           // dgdt = G'(T)
{
    const double kT = kB_300*T/300;

    Eex = Compute_Eex(T);
    double s1, s2 = gr[0], s3, s4 = 0., delta, tmp;

    for(int r=1; r<nb_item; ++r) {
        delta = dEA - dEr[r];
        tmp = gr[r]*exp( -delta/kT );
```

```
s2 += tmp;
s4 += tmp*delta/kT/T;           // s4 = ds2/dT
}

delta = dEA - Eex;
s1 = 4.*exp( delta/kT );
double dEEx_dT = Compute_dEExdT(T);
s3 = s1*( -dEEx_dT/kT - delta/kT/T ); // s3 = ds1/dT

dgdt = s3*s2 + s1*s4;          // dgdt = d(s1*s2)/dT

return;
}

// Eex is given by      Equation 1369 on page 1475
double Matsuura_DistributionFunction::Compute_Eex(double T)
{
    const double kT = kB_300*T/300;

    double s1 = 0., s2 = gr[0];

    for(int r=1; r<nb_item; ++r) {
        double delta = dEA - dEr[r];
        double tmp = gr[r]*exp( -delta/kT );
        s1 += delta*tmp;
        s2 += tmp;
    }

    return s1/s2;
}

double Matsuura_DistributionFunction::Compute_dEExdT(double T)
{
    const double kT = kB_300*T/300;

    double s1 = 0., s2 = gr[0], s3 = 0., s4 = 0.;

    for(int r=1; r<nb_item; ++r) {
        double delta = dEA - dEr[r];
        double tmp = gr[r]*exp( -delta/kT );
        s1 += delta*tmp;
        s2 += tmp;
        s3 += delta*tmp*delta/kT/T; // s3 = ds1/dT
        s4 += tmp*delta/kT/T;     // s4 = ds2/dT
    }

    return (s3*s2 - s1*s4)/s2/s2;
}

extern "C"
PMI_DistributionFunction* new_PMI_DistributionFunction
```

## Chapter 39: Physical Model Interface

Various

```
(const PMI_Environment& env,
  const char* name,
  const PMI_SpeciesType type)
{
    return new Matsuura_DistributionFunction (env, name, type);
}

void Compute_dEr(int nb_item, double* dEr)
{
    // dEr is given by Equation 1367 on page 1475

    // data from file: 6H-SiC.par
    const double epsilon = 9.66;           // dielectric constant
    const double mh      = 1;              // hole effective mass in SiC

    const double E0 = 13.6*mh/epsilon/epsilon;

    for(int r=1; r<=nb_item; ++r) {
        dEr[r-1] = E0/r/r;
    }
}
```

---

## Space Factor

The space distribution of the metal workfunction (see [Metal Workfunction on page 298](#)), traps (see [Energetic and Spatial Distribution of Traps on page 544](#)), the bond concentration (see [Using the Trap Degradation Model on page 606](#)), the extended nonradiative multiphonon (eNMP) model precursor concentration (see [Using the eNMP Model on page 628](#)), and piezoresistance enhancement factors (see [SFactor Dataset or PMI Model on page 994](#)) can be computed by a space factor PMI. The name of the PMI is specified in the appropriate Physics section as follows:

```
Physics (Material | Region = "<name>") {
    MetalWorkfunction (SFactor=pmi_model_name ...)
```

or:

```
Physics { Traps (SFactor=pmi_model_name ...) }
```

or:

```
Physics { Traps (BondConcSFactor=pmi_model_name ...) }
```

or:

```
Physics (MaterialInterface | RegionInterface = "<name1>/<name2>") {
    eNMP (SFactor=pmi_model_name ...)
```

## Chapter 39: Physical Model Interface

Various

or:

```
Physics {
    Piezo (
        Model (
            Mobility (
                Factor (
                    SFactor=pmi_model_name
                    [ChannelDirection=<n>]
                    [AutoOrientation | ParameterSetName=<psname> ]
                )
            )
        )
    )
}
```

In the last specification, the `Factor` options `ChannelDirection=<n>`, `AutoOrientation`, and `ParameterSetName=<psname>`, if specified, are passed as parameters to the space factor PMI model (`AutoOrientation` is passed as `AutoOrientation=1`).

### Note:

The name of the PMI model must not coincide with the name of an internal field of Sentaurus Device. Otherwise, Sentaurus Device takes the value of the internal field as the space factor.

## Dependencies

A PMI space factor model has no explicit dependencies. The model must compute:

`spacefactor` Space factor (1) or [ $\text{cm}^{-3}$ ]

## Standard C++ Interface

The following base class is declared in the file `PMIModels.h`:

```
class PMI_SpaceFactor : public PMI_Vertex_Interface {

public:
    PMI_SpaceFactor (const PMI_Environment& env);
    virtual ~PMI_SpaceFactor ();

    virtual void Compute_spacefactor
        (double& spacefactor) = 0;
};
```

The prototype for the virtual constructor is given as:

```
typedef PMI_SpaceFactor* new_PMI_SpaceFactor_func
    (const PMI_Environment& env);
extern "C" new_PMI_SpaceFactor_func new_PMI_SpaceFactor;
```

## Simplified C++ Interface

The following base class is declared in the file `PMI.h`:

```
class PMI_SpaceFactor_Base : public PMI_Vertex_Base {  
  
public:  
    class Input : public PMI_Vertex_Input_Base {  
    public:  
    };  
  
    class Output {  
    public:  
        pmi_float spacefactor; // space factor  
    };  
  
    PMI_SpaceFactor_Base (const PMI_Environment& env);  
    virtual ~PMI_SpaceFactor_Base ();  
  
    virtual void compute (const Input& input, Output& output) = 0;  
};
```

The prototype for the virtual constructor is given as:

```
typedef PMI_SpaceFactor_Base* new_PMI_SpaceFactor_Base_func  
    (const PMI_Environment& env);  
extern "C" new_PMI_SpaceFactor_Base_func new_PMI_SpaceFactor_Base;
```

## Example: PMI User Field as Space Factor

The following code reads the space factor from a PMI user field:

```
#include "PMIModels.h"  
  
class pmi_spacefactor : public PMI_SpaceFactor {  
  
public:  
    pmi_spacefactor (const PMI_Environment& env);  
    ~pmi_spacefactor ();  
  
    void Compute_spacefactor (double& spacefactor);  
};  
  
pmi_spacefactor::  
pmi_spacefactor (const PMI_Environment& env) :  
    PMI_SpaceFactor (env)  
{  
}  
  
pmi_spacefactor::  
~pmi_spacefactor ()  
{  
}
```

```
void pmi_spacefactor::  
Compute_spacefactor (double& spacefactor)  
{ spacefactor = ReadUserField (PMI_UserField1);  
}  
  
extern "C"  
PMI_SpaceFactor* new_PMI_SpaceFactor  
(const PMI_Environment& env)  
{ return new pmi_spacefactor (env);  
}
```

---

## References

- [1] H. Matsuura, “Influence of Excited States of Deep Acceptors on Hole Concentration in SiC,” in *International Conference on Silicon Carbide and Related Materials (ICSCRM)*, Tsukuba, Japan, pp. 679–682, October 2001.
- [2] P. Y. Yu and M. Cardona, *Fundamentals of Semiconductors: Physics and Materials Properties*, Berlin: Springer, 2nd ed., 1999.
- [3] K. F. Brennan, *The Physics of Semiconductors: With applications to optoelectronic devices*, Cambridge: Cambridge University Press, 1999.
- [4] M. G. Ancona, “Density-gradient theory: a macroscopic approach to quantum confinement and tunneling in semiconductor devices,” *Journal of Computational Electronics*, vol. 10, no. 1–2, pp. 65–97, 2011.
- [5] P. Bruno, “Interlayer Exchange Interactions in Magnetic Multilayers,” *Magnetism: Molecules to Materials III*, J. S. Miller and M. Drillon (eds.), Wiley-VCH: Weinheim, pp. 329–353, 2002.
- [6] M. D. Stiles, “Interlayer Exchange Coupling,” *Ultrathin Magnetic Structures III: Fundamentals of Nanomagnetism*, J. A. C. Bland and B. Heinrich (eds.), Springer: Berlin, pp. 99–142, 2005.

# 40

## Tcl Interfaces

---

*This chapter discusses the Tcl interfaces that can be used to customize device simulations in Sentaurus Device.*

---

### Overview

You can use Tcl scripts to control various aspects of a Sentaurus Device simulation. Tcl scripts can be used in the following circumstances:

- Use the Tcl interpreter to execute a command file (see [Tcl Command File on page 214](#)).
- Tcl expressions are recognized in the context of enhanced spectrum control to compute optical generation (see [Enhanced Spectrum Control on page 653](#)).
- Use a Tcl formula to add data to the current plot file (see [Tcl Formulas on page 170](#)).
- The Tcl current plot interface (see [Current Plot File on page 1491](#)) represents an alternative to the current plot PMI described in [Current Plot File on page 1439](#).

Mesh-based Tcl interfaces, such as the current plot interface, need access to Sentaurus Device mesh and data. This runtime support is described in [Mesh-Based Runtime Support](#).

---

### Mesh-Based Runtime Support

The Tcl runtime environment is accessed through a Tcl pointer `tcl_cp_addr`:

```
upvar #1 tcl_cp_addr tcl_cp_addr
```

When the Tcl pointer `tcl_cp_addr` to the runtime environment has been obtained, you can perform the following operations:

- Read a parameter from the command file:

```
$tcl_cp_addr InitParameter $name $defaultvalue
```

- Read a string parameter from the command file:

```
$tcl_cp_adr InitStringParameter $name $defaultvalue
```

- Read the time in seconds during transient simulations:

```
$tcl_cp_adr ReadTime
```

- Read the step size in seconds during transient simulations:

```
$tcl_cp_adr ReadTransientStepSize
```

- Read the step type during transient simulations:

```
$tcl_cp_adr ReadTransientStepType
```

Returns either `$::PMI_UndefStepType`, `$::PMI_TR`, `$::PMI_BDF`, or `$::PMI_BE`.

- Read pointer to the Sentaurus Device mesh (see [Device Mesh on page 1484](#)):

```
set mesh [$tcl_cp_adr Mesh]
```

- Read pointer to the Sentaurus Device data (see [Device Data on page 1489](#)):

```
set data [$tcl_cp_adr Data]
```

---

## Device Mesh

Use a pointer to the Sentaurus Device mesh, like `set mesh [$tcl_cp_adr Mesh]`, for the following operations:

- Dimension of mesh:

```
$mesh dim
```

- Read element  $(i, j)$  of reference coordinate system, where  $0 \leq i, j < \text{dim}$ :

```
$mesh ref_coordinates $i $j
```

- Number of vertices:

```
$mesh size_vertex
```

- Number of element vertices:

```
$mesh size_element_vertex
```

- Read pointer to vertex  $i$ :

```
set vertex [$mesh vertex $i]
```

- Number of edges:

```
$mesh size_edge
```

- Read pointer to edge  $i$ :  
`set edge [$mesh edge $i]`
- Number of elements:  
`$mesh size_element`
- Read pointer to element  $i$ :  
`set element [$mesh element $i]`
- Number of regions:  
`$mesh size_region`
- Read pointer to region  $i$ :  
`set region [$mesh region $i]`
- Number of region interfaces:  
`$mesh size_regioninterface`
- Read pointer to region interface  $i$ :  
`set regioninterface [$mesh regioninterface $i]`

## Vertex

Use a pointer to a vertex for the following operations:

- Read index to access vertex data:  
`$vertex index`
- Read index to access element–vertex data:  
`$vertex element_vertex_index $element`
- Read coordinates:  
`$vertex coord $d`  
The value of  $d$  determines the component,  $0 \leq d < \dim$ .
- Determine whether this vertex has the same coordinates as vertex  $v$ :  
`$vertex equal_coord $v`
- Number of edges connected to this vertex:  
`$vertex size_edge`

- Read pointer to edge *i*:

```
set edge [$vertex edge $i]
```

- Number of elements connected to this vertex:

```
$vertex size_element
```

- Read pointer to element *i*:

```
set element [$vertex element $i]
```

- Number of regions connected to this vertex:

```
$vertex size_region
```

- Read pointer to region *i*:

```
set region [$vertex region $i]
```

- Number of region interfaces connected to this vertex:

```
$vertex size_regioninterface
```

- Read pointer to region interface *i*:

```
set regioninterface [$vertex regioninterface $i]
```

## Edge

Use a pointer to an edge for the following operations:

- Read index to access edge data:

```
$edge index
```

- Read pointer to first vertex of edge:

```
set vertex [$edge start]
```

- Read pointer to second vertex of edge:

```
set vertex [$edge end]
```

- Number of elements connected to this edge:

```
$edge size_element
```

- Read pointer to element *i*:

```
set element [$edge element $i]
```

- Number of regions containing edge:

```
$edge size_region
```

- Read pointer to region  $i$ :

```
set region [$edge region $i]
```

## Element

Use a pointer to an element for the following operations:

- Read index to access element data:

```
$element index
```

- Read type of element:

```
$element type
```

Returns one of the following values:

```
$::des_element_point  
$::des_element_line  
$::des_element_triangle  
$::des_element_rectangle  
$::des_element_tetrahedron  
$::des_element_pyramid  
$::des_element_prism  
$::des_element_cuboid  
$::des_element_tetrabrick
```

- Number of vertices in element:

```
$element size_vertex
```

- Read pointer to vertex  $i$ :

```
set vertex [$element vertex $i]
```

- Number of edges connected to this element:

```
$element size_edge
```

- Read pointer to edge  $i$ :

```
set edge [$element edge $i]
```

- Read pointer to bulk region containing element:

```
set bulk [$element bulk]
```

- Start index for element–vertex data in element:

```
$element element_vertex_offset
```

## Region

Use a pointer to a region for the following operations:

- Read index:

```
$region index
```

- Read type of region:

```
$region type
```

Returns either `$::des_region_bulk` or `$::des_region_contact`.

- Read name of region:

```
$region name
```

- Number of vertices in region:

```
$region size_vertex
```

- Read pointer to vertex *i*:

```
set vertex [$region vertex $i]
```

- Number of edges in region:

```
$region size_edge
```

- Read pointer to edge *i*:

```
set edge [$region edge $i]
```

Use the following functions to convert a pointer to a region into a pointer to a bulk region or a contact:

```
set bulk [tcl_cp_region2bulk $region]
set contact [tcl_cp_region2contact $region]
```

A pointer to a bulk region supports the following additional operations:

- Read material of region:

```
$bulk material
```

- Number of elements in region:

```
$bulk size_element
```

- Rad pointer to element *i*:

```
set element [$bulk element $i]
```

- Number of region interfaces in region:  
`$bulk size_regioninterface`
- Read pointer to region interface *i*:  
`set regioninterface [$bulk regioninterface $i]`

## Region Interface

Use a pointer to a region interface for the following operations:

- Read index:  
`$regioninterface index`
- Read pointer to first bulk region connected to region interface:  
`set bulk [$regioninterface bulk1]`
- Read pointer to second bulk region connected to region interface:  
`set bulk [$regioninterface bulk2]`
- Determine whether this region interface is a heterointerface:  
`$regioninterface is_heterointerface`
- Number of vertices in region interface:  
`$regioninterface size_vertex`
- Read pointer to vertex *i*:  
`set vertex [$regioninterface vertex $i]`
- Read index to access data stored on region interface vertex *i*:  
`$regioninterface index $i`

---

## Device Data

Use a pointer to the Sentaurus Device data, like `set data [$tcl_cp_addr Data]`, for the following operations:

- Read pointer to box method coefficients (2D array, C++ data type `double**`):  
`set coefficient [$data ReadCoefficient]`
- Read pointer to box method measures (2D array, C++ data type `double**`):  
`set measure [$data ReadMeasure]`

## Chapter 40: Tcl Interfaces

### Mesh-Based Runtime Support

- Read pointer to box method surface measures (2D array, C++ data type `double**`):  

```
set surfacemeasure [$data ReadSurfaceMeasure]
```
- Read pointer to scalar data (1D array, C++ data type `double*`):  

```
set scalar [$data ReadScalar $location $name]
```
- Read pointer to vector data (2D array, C++ data type `double**`):  

```
set vector [$data ReadVector $location $name]
```
- Write scalar data (1D array, C++ data type `double*`):  

```
$data WriteScalar $location $name $newvalue
```
- Read pointer to gradient (2D array, C++ data type `double**`):  

```
set gradient [$data ReadGradient $location $name]
```
- Read pointer to flux (1D array, C++ data type `double*`):  

```
set flux [$data ReadFlux $location $name]
```
- Electron distribution from SHE method:  

```
$data ReadeSHEDistribution $bulk $vertex $energy
```
- Hole distribution from SHE method:  

```
$data ReadhSHEDistribution $bulk $vertex $energy
```
- Electron density-of-states from SHE method:  

```
$data ReadeSHETotalDOS $bulk $energy
```
- Hole density-of-states from SHE method:  

```
$data ReadhSHETotalDOS $bulk $energy
```
- Electron group velocity from SHE method:  

```
$data ReadeSHETotalGSV $bulk $energy
```
- Hole group velocity from SHE method:  

```
$data ReadhSHETotalGSV $bulk $energy
```

The following values are recognized for `$location`:

```
$::des_data_vertex  
$::des_data_edge  
$::des_data_element  
$::des_data_rivertex  
$::des_data_element_vertex
```

## Chapter 40: Tcl Interfaces

### Current Plot File

See [Appendix F on page 1515](#) for the names of scalar and vector data.

## One-Dimensional Arrays

Use the following function to allocate a 1D array (C++ data type `double*`):

```
set v1 [tcl_cp_new_double $size]
```

Use the following function to read an element of a 1D array (C++ data type `double*`):

```
set value [tcl_cp_get_double $v1 $index]
```

Use the following function to write an element of a 1D array (C++ data type `double*`):

```
tcl_cp_set_double $v1 $index $value
```

Use the following function to deallocate a 1D array (C++ data type `double*`):

```
tcl_cp_delete_double $v1
```

## Two-Dimensional Arrays

Use the following function to read an element of a 2D array (C++ data type `double**`):

```
set value [tcl_cp_get_double2 $v2 $index1 $index2]
```

---

## Current Plot File

You can use the current plot Tcl interface to add new entries to the current plot file. It is functionally equivalent to the current plot PMI described in [Current Plot File on page 1439](#).

The required Tcl code must be specified through a `tcl` statement in the `CurrentPlot` section. For example:

```
CurrentPlot {  
    Tcl (tcl = "source CurrentPlot.tcl"  par1=<value> par2=<value> )  
}
```

In this example, the Tcl code is stored in the file `CurrentPlot.tcl`. If necessary, you can specify the model parameters (`par1` and `par2`) as well.

---

## Tcl Functions

You must define Tcl functions for the following purposes:

- `tcl_cp_constructor`: Constructor
- `tcl_cp_destructor`: Destructor (optional)

## Chapter 40: Tcl Interfaces

### Current Plot File

- `tcl_cp_Compute_Dataset_Names`: Compute a list of dataset names for the header of the current plot file
- `tcl_cp_Compute_Function_Names`: Compute a list of function names for the header of the current plot file
- `tcl_cp_Compute_Plot_Values`: Evaluate the current plot values

These Tcl procedures must implement the same functionality as the corresponding methods of the C++ class `PMI_CurrentPlot` (see [Current Plot File on page 1439](#)). The Tcl interpreter also has access to runtime support functions and the entire Sentaurus Device mesh and data fields (see [Mesh-Based Runtime Support on page 1483](#)).

### **tcl\_cp\_constructor**

The constructor is invoked once at the beginning of each Tcl current plot statement:

```
proc tcl_cp_constructor {} {
    upvar #1 tcl_cp_addr tcl_cp_addr
    ...
}
```

All the current plot Tcl procedures are executed in their own Tcl namespace. This ensures that multiple Tcl current plot statements can be active simultaneously, and they all operate in their own private namespace. Use the Tcl `upvar` command to access variables in this namespace.

As explained in [Mesh-Based Runtime Support on page 1483](#), the Tcl pointer `tcl_cp_addr` provides access to the Sentaurus Device mesh and data:

```
set mesh [$tcl_cp_addr Mesh]
set data [$tcl_cp_addr Data]
```

The constructor also can be used to precompute data that will be needed later to evaluate the current plot values.

### **tcl\_cp\_destructor**

This optional procedure can be used to deallocate data structures, or to print statistical output:

```
proc tcl_cp_destructor {} {
```

### **tcl\_cp\_Compute\_Dataset\_Names**

This procedure must return a Tcl list of dataset names. For example:

```
proc tcl_cp_Compute_Dataset_Names {} {
    lappend result "channel eConductivity"
```

## Chapter 40: Tcl Interfaces

### Current Plot File

```
        return $result
    }
```

Multiple dataset names are supported.

## tcl\_cp\_Compute\_Function\_Names

This procedure must return a Tcl list of function names. For example:

```
proc tcl_cp_Compute_Function_Names {} {
    lappend result "Conductivity"
    return $result
}
```

The number of function names must be identical to the number of dataset names created by tcl\_cp\_Compute\_Dataset\_Names.

## tcl\_cp\_Compute\_Plot\_Values

This procedure must return a Tcl list of current plot values. Use the access functions to the Sentaurus Device mesh and data (see [Mesh-Based Runtime Support on page 1483](#)) to compute these values:

```
proc tcl_cp_Compute_Plot_Values {} {
    ...
    lappend result 0.0
    return $result
}
```

The number of result values must be identical to the number of dataset names created by tcl\_cp\_Compute\_Dataset\_Names.

---

## Example: Average Electron Conductivity

This example computes the average of the electron conductivity  $\sigma_n = qn\mu_n$  in a region and is located in the directory \$STROOT/tcad/\$STRELEASE/lib/sdevice/src/tcl\_currentplot:

```
proc tcl_cp_constructor {} {
    # link to variables in enclosing namespace
    upvar #1 tcl_cp_adr tcl_cp_adr
    upvar #1 Conductivity_Region Conductivity_Region

    set Conductivity_Region \
        [$tcl_cp_adr InitStringParameter "Conductivity_Region" "" ]
}

proc tcl_cp_destructor {} {
```

## Chapter 40: Tcl Interfaces

### Current Plot File

```
proc tcl_cp_Compute_Dataset_Names {} {
    upvar #1 Conductivity_Region Conductivity_Region
    lappend result "Tcl_Ave_$Conductivity_Region eConductivity"
    return $result
}

proc tcl_cp_Compute_Function_Names {} {
    lappend result "Conductivity"
    return $result
}

proc tcl_cp_Compute_Plot_Values {} {
    # link to variables in enclosing namespace
    upvar #1 tcl_cp_addr tcl_cp_addr
    upvar #1 Conductivity_Region Conductivity_Region

    set mesh [$tcl_cp_addr Mesh]
    set data [$tcl_cp_addr Data]
    set measure [$data ReadMeasure]

    set q 1.602e-19
    set eDensity [$data ReadScalar $::des_data_vertex "eDensity"]
    set eMobility [$data ReadScalar $::des_data_vertex "eMobility"]

    set sum 0.0
    set sum_m 0.0

    set size_region [$mesh size_region]
    for {set ri 0} {$ri < $size_region} {incr ri} {
        set region [$mesh region $ri]
        if {[${region type}] == $::des_region_bulk && \
            [${region name}] == $Conductivity_Region} {
            set bulk [tcl_cp_region2bulk $region]
            set size_element [$bulk size_element]
            for {set ei 0} {$ei < $size_element} {incr ei} {
                set element [$bulk element $ei]
                set element_index [$element index]
                set size_vertex [$element size_vertex]
                for {set vi 0} {$vi < $size_vertex} {incr vi} {
                    set vertex [$element vertex $vi]
                    set vertex_index [$vertex index]

                    set n [tcl_cp_get_double $eDensity $vertex_index]
                    set mu [tcl_cp_get_double $eMobility $vertex_index]
                    set value [expr "$q * $n * $mu"]

                    set m [tcl_cp_get_double2 $measure $element_index $vi]
                    set sum [expr "$sum + $m * $value"]
                    set sum_m [expr "$sum_m + $m"]
                }
            }
        }
    }
}
```

## Chapter 40: Tcl Interfaces

### Current Plot File

```
        }

if {$sum_m == 0} {
    set average 0
} else {
    set average [expr "$sum / $sum_m"]
}

lappend result $average
}
```

# 41

## Python Interface

---

*This chapter discusses the Python interface that can be used to create customized device simulation flows.*

---

### Overview of Python Interface

The simulation capabilities of Sentaurus Device are available as a Python module that can be loaded into the Python interpreter gpythonsh (see *Sentaurus™ Workbench User Guide*, gpythonsh). You can use the Python interface to create advanced simulation flows that can utilize the multitude of available Python packages and frameworks.

The Sentaurus Device Python interface consists of the class `SDevice`, which provides the core simulation steps, and the class `CurrentPlotManager`, which gives in-memory access to data usually available in current plot (`.plt`) files. After importing the module `sdevice`, you can access documentation about the classes by using the `help()` method in Python as usual:

```
from sdevice import *
help(SDevice)
help(CurrentPlotManager)
```

The Python interface can be used interactively or in batch mode by passing a Python script with Sentaurus Device commands to the Python interpreter:

```
gpythonsh <Sentaurus_Device_Python_command_script>
```

For a richer experience, you also can use the Python interface interactively with IPython, which is distributed as part of the general-purpose Python shell:

```
gpythonsh -m IPython
```

## Performing Device Simulations

The class `SDevice` provides all the core methods to perform a device simulation. The arguments to the methods expect a string containing the respective standard Sentaurus Device command file syntax.

A device simulation consists of an initialization step followed by the solving of the equations, which allows for several `Solve` statements with intermediate stages with general Python code. The latter can be used to configure the next `Solve` statement based on the results of the previous one. For example:

```
from sdevice import *

sd = SDevice()
sd.init('''<all Sentaurus Device command file content except Solve
          section>''')
sd.solve('''<Sentaurus Device command Solve section>''')
... # other Python code
sd.solve('''<Sentaurus Device command Solve section>''')
...
sd.finish()           # clean up Sentaurus Device simulation and close
                      # all plt files
```

---

## Accessing Current Plot Data

You can use the class `CurrentPlotManager` to access data usually available in current plot (`.plt`) files. You can query all available plot (`.plt`) files of a simulation and then request information about a specific plot, such as available variables and their units, to obtain the respective value arrays of interest.

**Note:**

Sentaurus Device can create multiple current plot files. Examples are in mixed-mode simulations, in which multiple devices are used in a `System` section, or the use of `NewCurrentPrefix` or `.plt` files generated by specific models.

After the device simulation has been initialized using the `init()` method, you can obtain a `CurrentPlotManager` object:

```
cpm = sd.get_current_plot_manager()
```

After calling the `solve()` method, you can request a list of all current plots:

```
cplots = cpm.current_plots()
```

## Chapter 41: Python Interface

### Accessing Current Plot Data

For a specific current plot, you can query the available plot variables:

```
print(cplots)
("n1_des.plt")
variables = cpm.get_current_plot_variables(cplots[0])
# retrieve all variables of the first current plot
```

The variable `variables` contains a list of tuples, where each tuple represents a variable by holding its variable name, function name, and unit:

```
print(variables[0])
('drain OuterVoltage', 'OuterVoltage', 'V')
```

To access all current plot values for a specific current plot, use:

```
vals = cpm.get_current_plot_values(cplots[0])
```

Alternatively, to access only the values for a subset of variables, use:

```
vals = cpm.get_current_plot_values(cplots[0], [x[0] for x in
variables[2:4]])
```

The second argument, if provided, must be a list of valid variable names as obtained from the call to `get_current_plot_variables()`. For example:

```
vd = cpm.get_current_plot_values(cplots[0], ['drain OuterVoltage'])
```

If you are not interested in any current plot values or are interested only in the values from the latest simulation step, then you can set the data range accordingly:

```
cpm.set_data_range(cpm.data_range.none)
```

or:

```
cpm.set_data_range(cpm.data_range.latest)
```

A corresponding method is also available to query the current settings:

```
cpm.get_data_range()
```

The method `get_current_plot_values()` returns an *N*-dimension array type, `ndarray`, from the Python package NumPy. Each array holds the values for a specific variable according to the input list of variables.

#### Note:

Current plot values are read-only and any changes are not reflected in `.plt` files written to disk by Sentaurus Device.

## Limitations

The following limitations apply to the Python interface:

- The creation of more than one `SDevice` object in the same Python interpreter is not supported.
- If there is a `Set(TrapFilling=...)` statement inside an `SDevice.solve()` method, then it applies only to subsequent statements within the surrounding `Solve` section (see [Explicit Trap Occupation on page 550](#)). This effect does not persist between multiple calls of the `SDevice.solve()` method.
- If a transient simulation is performed using several `SDevice.solve()` methods, then the correct initial time, that is, the final time of the last transient statement, must be specified because Sentaurus Device does not retain the last transient time from the previous `SDevice.solve()` method.

## **Part IV: Appendices**

---

This part of the *Sentaurus™ Device User Guide* contains the following appendices:

- [Appendix A, Mathematical Symbols](#)
- [Appendix B, Syntax](#)
- [Appendix C, File-Naming Conventions](#)
- [Appendix D, Command-Line Options](#)
- [Appendix E, Runtime Statistics](#)
- [Appendix F, Data Names and Plot Names](#)
- [Appendix G, Command File Overview](#)

# A

## Mathematical Symbols

---

*This appendix contains notational conventions and a list of symbols used in the Sentaurus™ Device User Guide.*

They are listed alphabetically. Non-Latin characters are sorted according to their English translation.

/	Division, right binding: $a/bc = a/(bc)$
$\hat{a}$	Unit (3D) vector
$\vec{a}$	(3D) vector
$ a $	Absolute value of a scalar
$ \vec{a} $	Absolute value of a vector $a =  \vec{a} $
$a^T$	Transpose of a vector or a matrix
$a^{-1}$	Inverse of function or matrix, reciprocal of a scalar
$\vec{a} \cdot \vec{b}$	Inner (dot) product
$\vec{a} \times \vec{b}$	Vector (cross) product
$\vec{a}\vec{b}$	Dyadic product: $(\vec{a}\vec{b})_{ij} = \vec{a}_i \vec{b}_j$

## Appendix A: Mathematical Symbols

$abc\bar{c}$	Miller indices. $a$ , $b$ , and $c$ are digits; a bar over a digit indicates negation applied to that particular digit.
$\chi$	Electron affinity or thermal resistivity
$c_L$	Lattice heat capacity
$e$	Base of natural logarithm
$\overset{\rightarrow}{E}$	Electric field
$E_{\text{bgn}}$	Bandgap narrowing
$E_C$	Conduction band energy
$E_{F,n}$	Electron quasi-Fermi energy
$E_{F,p}$	Hole quasi-Fermi energy
$E_g$	Intrinsic band gap
$E_{g,\text{eff}}$	Effective band gap, $E_{g,\text{eff}} = E_g - E_{\text{bgn}}$
$E_V$	Valence band energy
$\epsilon$	Absolute dielectric constant of a material
$\epsilon_0$	Dielectric constant of vacuum
$\overset{\rightarrow}{F}$	Electric field
$F_\alpha$	Integral of distribution function; for Fermi statistics, Fermi integral of order $\alpha$
$G$	Generation rate (does not include recombination)
$\gamma_n$	Degeneracy factor for electrons

## Appendix A: Mathematical Symbols

$\gamma_p$	Degeneracy factor for holes
$\hbar$	Planck's constant divided by $2\pi$
$i$	Imaginary unit, $i^2 = -1$
$Im$	Imaginary part
$\vec{J}_D$	Displacement current density
$\vec{J}_M$	Current density in metals
$\vec{J}_n$	Electron current density
$\vec{J}_p$	Hole current density
$k$	Boltzmann constant
$\kappa_L$	Lattice thermal conductivity
$\kappa_n$	Electron thermal conductivity
$\kappa_p$	Hole thermal conductivity
$\Lambda_n$	Electron quantum potential
$\Lambda_p$	Hole quantum potential
$ln$	Natural logarithm
$m_0$	Free electron mass
$m_n$	Electron density-of-states mass
$m_p$	Hole density-of-states mass

## Appendix A: Mathematical Symbols

$\mu_n$	Electron mobility
$\mu_p$	Hole mobility
$n$	Electron density
$\hat{n}$	Unit normal vector
$N_{A,0}$	Chemically active acceptor concentration
$N_A$	Ionized acceptor concentration
$N_C$	Conduction band density-of-states
$N_{D,0}$	Chemically active donor concentration
$N_D$	Ionized donor concentration
$n_i$	Intrinsic density (not accounting for bandgap narrowing)
$n_{i,eff}$	Effective intrinsic density (accounting for bandgap narrowing)
$N_i$	Ionized dopant concentration, $N_i = N_A + N_D$
$n_{se}$	Single exciton density
$N_{tot}$	Total doping concentration, $N_{tot} = N_{A,0} + N_{D,0}$
$N_V$	Valence band density-of-states
$p$	Hole density
$\vec{P}$	Polarization
$P_n$	Electron thermoelectric power

## Appendix A: Mathematical Symbols

$P_p$	Hole thermoelectric power
$\Phi_M$	Metal Fermi potential
$\Phi_n$	Electron quasi-Fermi potential
$\Phi_p$	Hole quasi-Fermi potential
$\phi$	Electrostatic potential
$q$	Elementary charge
$r$	Anisotropy factor
$R$	Recombination rate (does not include generation)
$R_{\text{net}}$	Net recombination rate, $R_{\text{net}} = R - G$
$\text{Re}$	Real part
$\vec{S}_L$	Lattice heat flux density
$\vec{S}_n$	Electron heat flux density
$\vec{S}_p$	Hole heat flux density
$T$	Lattice temperature
$T_n$	Electron temperature
$T_p$	Hole temperature
$\tau_n$	Electron lifetime
$\tau_p$	Hole lifetime

## Appendix A: Mathematical Symbols

$\Theta$	Unit step function (0 for negative arguments, 1 for positive arguments)
$\triangleright v_n$	Electron drift velocity
$\triangleright v_p$	Hole drift velocity
$\triangleright v_{\text{sat},n}$	Electron saturation velocity
$\triangleright v_{\text{sat},p}$	Hole saturation velocity

# B

## Syntax

---

*The syntax of the command file of Sentaurus Device, and the basic syntactical and lexical conventions are described here.*

Sentaurus Device has a hierarchical input syntax. At the lowest level, device, system, and solve information is specified as well as the default and global parameters. Inside each Device section, the parameters specific to one device type can be specified.

Inside the System section, the real devices are specified or instantiated. Here, parameters can be given that are specific to one instantiation of a device. The command file is a collection of specifications used to establish the simulation environment with actions describing which equations must be solved and how they must be solved. The syntax of the command file contains several entry types. All basic command file entries adhere to the syntactical and lexical rules described in [Table 193](#).

*Table 193 Entry types in Sentaurus Device*

Entry type	Description
Keyword	These are the known names of the command file. They are case insensitive. Therefore, the following keywords are all equivalent: Quasistationary, QuasiStationary, and quasistationary. Most keywords can be abbreviated. This example can also be written as quasiStat.
Integer	These are (possibly) signed decimal numbers. The following integers are valid: 123, -73492, 0
Float	Floating-point numbers are compatible with the C language format for floating-point numbers. The following floating-point numbers are valid: 123, 123.0, 1.23e2, -1.23E2

## Appendix B: Syntax

*Table 193 Entry types in Sentaurus Device (Continued)*

Entry type	Description
Vector	Vectors in real space are defined depending on the actual dimension as follows: <ul style="list-style-type: none"> <li>• In three dimensions, a vector is specified by three floating-point numbers.</li> <li>• In two dimensions, a vector is specified by two floating-point numbers enclosed in parentheses. The floating-point numbers are separated by commas or space.</li> <li>• In one dimension, one floating-point number without parentheses is sufficient.</li> </ul> Valid vectors are (1,0,2), (1e-4,-1e-3), and 1.
String	Strings are delimited by quotation marks. They are compatible with the C language format for strings. The following strings are valid: "Vdd", "output/diode"
Identifier	These are used to name objects such as nodes, devices, or attributes. They are compatible with the C language format for identifiers. The following identifiers are valid: vdd, diode, bjt_345
Assignment	These are used to set values to keywords. Therefore, the following are valid assignments: Digits=4, Save="output/diode"
Signal	Signals are time dependent, piecewise, linear functions (not to be confused with UNIX signals) that are defined as inputs on the contacts of a device. They are specified as follows: (value0 at time0, value1 at time1, ...value_n at time_n). The following signal is valid: (0 at 0, 1 at 10.0e-9, 1 at 20.0e-9)
List	Lists are collections of keywords, assignments, and complex entries. They are delimited by "(...)" or "{...} ". The following lists are valid: <pre>{ Number=0 Voltage=0     Voltage=( 0 at 0, 0 at 2e-8 ) } { Method=Super Digits=6 Numerically } ( MinStep=1e-15 InitialStep=1e-10 Digits=3 )</pre>
Structured entries	These are parameterized definitions or commands that can have the forms <keyword> {<keywords>}, <keyword> (<keywords>), or <keyword> (<list>) {<list>}.

# C

## File-Naming Conventions

---

*This appendix describes the file-naming conventions for TCAD tools relevant to Sentaurus Device.*

---

### File Extensions

All strings that represent file names containing a dot (.) within their base name are taken literally. Otherwise, Sentaurus Device extends the given strings with the appropriate extension.

Sentaurus Device expands the extensions for output files by its tool extension \_des, for example, the extension of a saved file is \_des.sav.

During transient, quasistationary, and continuation simulations, the plot and save files are numbered by a global index.

*Table 194 Summary of file extensions used in Sentaurus Device*

File	I/O	Extension
Command	I	_des.cmd, .cmd
Log	O	_des.log
Parameter	I	.par
Geometry/Doping	I	.tdr
Lifetime	I	.tdr
Save	O	_des.sav
Load	I	_des.sav
Device Plot (grid-based)	O	_des.tdr

## Appendix C: File-Naming Conventions

### File Extensions

Table 194    *Summary of file extensions used in Sentaurus Device (Continued)*

File	I/O	Extension
Current Plot	O	_des.plt
AC Extraction	O	_ac_des.plt
Montecarlo	I/O	See the <i>Sentaurus™ Device Monte Carlo User Guide</i> for more information

# D

## Command-Line Options

---

*This appendix lists the most useful command-line options available in Sentaurus Device.*

---

### Starting Sentaurus Device

To start Sentaurus Device, enter:

```
sdevice [<options>] [<commandfile>]
```

Sentaurus Device appends automatically the corresponding extension to the given command file if necessary. If no command file is specified, Sentaurus Device reads from standard input.

---

### Command-Line Options

Sentaurus Device interprets the following options:

Option	Description
-d	Prints debug information into the <code>debug</code> file. The information printed includes the numeric values of the Jacobian and RHS for each equation at each solution step.
-h	Lists these options and exits.
-i	Prints the initial solution in the save file, and print files specified in the <code>File</code> section of the command file, and exits without performing further computations.
-L	Writes the silicon model parameters into the file <code>Silicon.par</code> and exits.
-L <commandfile>	Writes model parameter files for all the materials, material interfaces, and electrodes used in <commandfile> and exits.

## Appendix D: Command-Line Options

### Command-Line Options

Option	Description
-L:<Material>	Writes a model parameter file <Material>.par for the specified material and exits.
-L:<Material>:<x>	Writes the model parameters for the given material and mole fraction into a file <Material>.par and exits.
-L:<Material>:<x>:<y>	Writes the model parameters for the given material and mole fractions into a file <Material>.par and exits.
-L:<Material>/<Material>	Writes a model parameter file <Material>%<Material>.par for the specified material interface and exits.
-L:All	Writes a separate model parameter file for all materials and exits.
-M <commandfile>	Writes a parameter file models-M.par for regions with computed mole fraction dependencies.
-n	Does not include Newton information in the log file.
-P	Writes the silicon model parameters into a file models.par and exits. This file can be modified and reloaded into Sentaurus Device to make customized changes to physical models and parameters.
-P <commandfile>	Writes the model parameters for the materials and interfaces used in <commandfile> into a file models.par and exits.
-P:<Material>	Writes the model parameters for the given material into a file models.par and exits.
-P:<Material>:<x>	Writes the model parameters for the given material and mole fraction into a file models.par and exits.
-P:<Material>:<x>:<y>	Writes the model parameters for the given material and mole fractions into a file models.par and exits.
-P:<Material>/<Material>	Writes the model parameters for the given material interface into a file models.par and exits.
-P:All	Writes the model parameters for all materials into a file models.par and exits.
-q	Quiet mode for output.
-r	When used with -L or -P, reads parameters from the material library to generate output.

## Appendix D: Command-Line Options

### Command-Line Options

Option	Description
-S	Writes the SiC model parameters into a <code>models_SiC.par</code> file and exits. This file can be modified and reloaded into Sentaurus Device to make customized changes to physical models and parameters.
-v	Prints header with version number of Sentaurus Device.
--compiler-version	Prints the version of the C++ compiler that was used to compile Sentaurus Device.
--exit-on-failure	Terminates immediately after a failed solve command.
--field-names	Prints fields and their numeric indices for use in the PMI.
--max_threads <int>	Limits the maximum number of threads used by the solver and assembly process.
--parameter-names	Prints the names of the parameters from the parameter file that can be ramped. If a command file is also supplied, Sentaurus Device prints the parameters from the command file that can be ramped.
--tcl	Invokes the Tcl interpreter to evaluate the command file.
--threads <int>	Sets the number of solver and assembly threads.
--verbose	Prints additional diagnostic messages (alternatively, set the environment variable <code>SDEVICE_VERBOSITY</code> to <code>high</code> ).
--xml	Creates an additional log file with XML tags. The file uses the extension <code>.xml</code> .

# E

## Runtime Statistics

---

*This appendix presents information about obtaining runtime statistics from Sentaurus Device.*

---

### Generating Statistics

The command `sdevicestat` displays some statistics of a previous run of Sentaurus Device based on the information found in its `log` file. For example, the command:

```
sdevicestat test_des.log
```

generates the following statistics:

Total number of Newton iterations	: 8
Number of restarts	: 0
Rhs-time	: 9.19 % ( 38.80 s )
Jacobian-time	: 1.43 % ( 6.05 s )
Solve-time	: 88.76 % ( 374.70 s )
Overhead	: 0.62 % ( 2.61 s )
Total CPU time (sum of above times)	: 422.16 s

The `sdevicestat` command recognizes whether the `WallClock` keyword has been specified in the `Math` section of the Sentaurus Device command file (see [Parallelization on page 221](#)). In this case, the same simulation on a dual processor machine produces the following output:

Total number of Newton iterations	: 8
Number of restarts	: 0
Rhs-time	: 8.57 % ( 20.68 s )
Jacobian-time	: 1.96 % ( 4.73 s )
Solve-time	: 88.33 % ( 213.07 s )
Overhead	: 1.14 % ( 2.74 s )
Total wallclock time (sum of above times):	241.22 s

# F

## Data Names and Plot Names

---

*This appendix provides information about data names and plot names.*

---

### Overview

[Table 195](#), [Table 196](#), [Table 197](#), and [Table 198](#) list the data names as they appear in the TDR and PLT files, and that are available in the current plot PMI (see [Current Plot File on page 1439](#)).

The plot names that are recognized in a `Plot` or `CurrentPlot` section of Sentaurus Device are identical to the data names if not mentioned otherwise (see [Device Plots on page 177](#)).

Vector data can be plotted by appending `/vector` to the corresponding keyword. For example:

```
Plot { ElectricField/Vector }
```

Element-based scalar data can be plotted by appending `/Element` to the corresponding keyword. For example:

```
Plot { eMobility/Element }
```

Special vector data can be plotted by appending `/SpecialVector` to the corresponding keyword. For example:

```
Plot { eSHEDistribution/SpecialVector }
```

Tensor data can be plotted by appending `/Tensor` to the corresponding keyword. For example:

```
Plot { Stress/Tensor }
```

**Note:**

The location *rivertex* refers to region-interface vertices.

## Appendix F: Data Names and Plot Names

### Scalar Data

For the kinetic Monte Carlo metal–insulator–metal (MIM) transport capability, [Table 199](#) lists the plot name that is recognized in the `Plot` section of a Sentaurus Device command file for saving particle data related to MIM defects.

---

## Scalar Data

*Table 195 Scalar data*

Data name	Location	Description	Unit
AbsorbedPhotonDensity	vertex	<a href="#">Quantum Yield Models on page 658</a>	$\text{cm}^{-3} \text{s}^{-1}$
AbsorbedPhotonDensityCoherent (Plot name: AbsorbedPhotonDensity)	vertex	<a href="#">Transfer Matrix Method on page 734</a>	$\text{cm}^{-3} \text{s}^{-1}$
AbsorbedPhotonDensityFromMonochromatic Source (Plot name: AbsorbedPhotonDensity)	vertex	<a href="#">Quantum Yield Models on page 658</a>	$\text{cm}^{-3} \text{s}^{-1}$
AbsorbedPhotonDensityFromSpectrum (Plot name: AbsorbedPhotonDensity)	vertex	<a href="#">Quantum Yield Models on page 658</a>	$\text{cm}^{-3} \text{s}^{-1}$
AbsorbedPhotonDensityIncoherent (Plot name: AbsorbedPhotonDensity)	vertex	<a href="#">Transfer Matrix Method on page 734</a>	$\text{cm}^{-3} \text{s}^{-1}$
AccepMinusConcentration	vertex	<a href="#">Specifying Doping Species on page 60</a>	$\text{cm}^{-3}$
AcceptorConcentration	vertex	<a href="#">Specifying Doping Species on page 60</a>	$\text{cm}^{-3}$
ActiveDopingConcentration	vertex	<a href="#">Specifying Doping Species on page 60, Using the Incomplete Ionization Model on page 339</a>	$\text{cm}^{-3}$
AlphaChargeDensity	vertex	<a href="#">Carrier Generation by Alpha Particles on page 773</a>	$\text{cm}^{-3}$
AlphaGeneration	vertex	$G^{\text{Alpha}}$ , <a href="#">Equation 777</a>	$\text{cm}^{-3} \text{s}^{-1}$

## Appendix F: Data Names and Plot Names

### Scalar Data

*Table 195 Scalar data (Continued)*

Data name	Location	Description	Unit
AntimonyActiveConcentration	vertex	<a href="#">Specifying Doping Species on page 60</a>	cm <sup>-3</sup>
AntimonyConcentration	vertex	<a href="#">Sb, Specifying Doping Species on page 60</a>	cm <sup>-3</sup>
AntimonyPlusConcentration	vertex	<a href="#">Sb+, Chapter 13 on page 339</a>	cm <sup>-3</sup>
ArsenicActiveConcentration	vertex	<a href="#">Specifying Doping Species on page 60</a>	cm <sup>-3</sup>
ArsenicConcentration	vertex	<a href="#">As, Specifying Doping Species on page 60</a>	cm <sup>-3</sup>
ArsenicPlusConcentration	vertex	<a href="#">As+, Chapter 13 on page 339</a>	cm <sup>-3</sup>
AugerRecombination	vertex	$R^A$ , <a href="#">Equation 441</a>	cm <sup>-3</sup> s <sup>-1</sup>
AutoOrientationSmoothing	vertex	<a href="#">Auto-Orientation Framework on page 86</a>	1
AvalFlatElementMax	element	<a href="#">Using Avalanche Generation on page 495</a>	degree
AvalFlatElementMin	element		degree
Band2BandGeneration	vertex	$G_{\text{net}}^{\text{bb}}$ , <a href="#">Band-to-Band Tunneling Models on page 523</a>	cm <sup>-3</sup> s <sup>-1</sup>
BandGap	vertex	$E_g$ , <a href="#">Bandgap and Electron-Affinity Models on page 305</a>	eV
BandgapNarrowing	vertex	$E_{\text{bgn}}$ , <a href="#">Bandgap and Electron-Affinity Models on page 305</a>	eV

## Appendix F: Data Names and Plot Names

### Scalar Data

*Table 195 Scalar data (Continued)*

Data name	Location	Description	Unit
BM_AngleElements	element	<a href="#">Statistics About Non-Delaunay Elements on page 1189</a>	degree
BM_AngleVertex	vertex		degree
BM_CoeffIntersectionNonDelaunayElements	element		1
BM_EdgesPerVertex	vertex		1
BM_ElementsPerVertex	vertex		1
BM_ElementsWithCommonObtuseFace	element		1
BM_ElementsWithObtuseFaceOnBoundaryDevice	element		1
BM_ElementVolume	element		$\mu\text{m}^3$
BM_IntersectionNonDelaunayElements	element		$\mu\text{m}$
BM_ShortestEdge	vertex		$\mu\text{m}$
BM_VolumeIntersectionNonDelaunayElements	element		$\mu\text{m}^3$
BM_wCoeffIntersectionNonDelaunayElements	element		1
BM_wElementsWithCommonObtuseFace	element		1
BM_wElementsWithObtuseFaceOnBoundaryDevice	element		1
BM_wIntersectionNonDelaunayElements	element		$\mu\text{m}$
BM_wVolumeIntersectionNonDelaunayElements	element		$\mu\text{m}^3$
BoronActiveConcentration	vertex	<a href="#">Specifying Doping Species on page 60</a>	$\text{cm}^{-3}$
BoronConcentration	vertex	<a href="#">B, Specifying Doping Species on page 60</a>	$\text{cm}^{-3}$
BoronMinusConcentration	vertex	<a href="#">B-, Chapter 13 on page 339</a>	$\text{cm}^{-3}$
BuiltinPotential	vertex	$\phi_0$ , <a href="#">Equation 101</a>	V

## Appendix F: Data Names and Plot Names

### Scalar Data

*Table 195 Scalar data (Continued)*

Data name	Location	Description	Unit
CBBARRIERLowering	vertex	Conduction band barrier lowering	eV
CBEnergywithLowering	vertex	Conduction band energy modified by barrier lowering	eV
CDL1Recombination	vertex	$R_1$ , <a href="#">Coupled Defect Level Recombination on page 487</a>	$\text{cm}^{-3} \text{s}^{-1}$
CDL2Recombination	vertex	$R_2$ , <a href="#">Coupled Defect Level Recombination on page 487</a>	$\text{cm}^{-3} \text{s}^{-1}$
CDLcRecombination	vertex	$R - R_1 - R_2$ , <a href="#">Coupled Defect Level Recombination on page 487</a>	$\text{cm}^{-3} \text{s}^{-1}$
CDLRecombination	vertex	$R$ , <a href="#">Coupled Defect Level Recombination on page 487</a>	$\text{cm}^{-3} \text{s}^{-1}$
ConductionBandEnergy	vertex, element-vertex	$E_C$ , <a href="#">Equation 41</a>	eV
ConductionCurrentDensity	vertex	$  \vec{J}_n + \vec{J}_p  $ , <a href="#">Equation 57</a> or $  \vec{J}_M  $ in metals, <a href="#">Equation 142</a>	$\text{Acm}^{-2}$

## Appendix F: Data Names and Plot Names

### Scalar Data

*Table 195 Scalar data (Continued)*

Data name	Location	Description	Unit
ConversePiezoelectricFieldxx	vertex	Components of converse piezoelectric field tensor	1
ConversePiezoelectricFieldxy			
ConversePiezoelectricFieldxz			
ConversePiezoelectricFieldyy			
ConversePiezoelectricFieldyz			
ConversePiezoelectricFieldzz			
cplxExtCoeff (Plot name: ComplexRefractiveIndex)	vertex	Complex Refractive Index Model on page 694	1
cplxRefIndex (Plot name: ComplexRefractiveIndex)	vertex	Complex Refractive Index Model on page 694	1
CurECImACGreenFunction	vertex	<a href="#">Table 142 on page 816</a>	1
CurECReACGreenFunction	vertex	<a href="#">Table 142 on page 816</a>	1
CurETIMACGreenFunction	vertex	<a href="#">Table 142 on page 816</a>	V <sup>-1</sup>
CurETReACGreenFunction	vertex	<a href="#">Table 142 on page 816</a>	V <sup>-1</sup>
CurGeoGreenFunction	vertex	<a href="#">Table 142 on page 816</a>	A cm <sup>-3</sup>
CurHCImACGreenFunction	vertex	<a href="#">Table 142 on page 816</a>	1
CurHCreACGreenFunction	vertex	<a href="#">Table 142 on page 816</a>	1
CurHTIMACGreenFunction	vertex	<a href="#">Table 142 on page 816</a>	V <sup>-1</sup>
CurHTReACGreenFunction	vertex	<a href="#">Table 142 on page 816</a>	V <sup>-1</sup>
CurLTIMACGreenFunction	vertex	<a href="#">Table 142 on page 816</a>	V <sup>-1</sup>
CurLTReACGreenFunction	vertex	<a href="#">Table 142 on page 816</a>	V <sup>-1</sup>

## Appendix F: Data Names and Plot Names

### Scalar Data

Table 195    *Scalar data (Continued)*

Data name	Location	Description	Unit
CurPotImACGreenFunction	vertex	<a href="#">Table 142 on page 816</a>	s <sup>-1</sup>
CurPotReACGreenFunction	vertex	<a href="#">Table 142 on page 816</a>	s <sup>-1</sup>
CurrentPotential	vertex	$W$ , Current Potential on page 242	Acm <sup>-1</sup>
d_k_carr (Plot name: ComplexRefractiveIndex)	vertex	Complex Refractive Index Model on page 694	1
d_k_lambda (Plot name: ComplexRefractiveIndex)	vertex	Complex Refractive Index Model on page 694	1
d_n_carr (Plot name: ComplexRefractiveIndex)	vertex	Complex Refractive Index Model on page 694	1
d_n_gain (Plot name: ComplexRefractiveIndex)	vertex	Complex Refractive Index Model on page 694	1
d_n_lambda (Plot name: ComplexRefractiveIndex)	vertex	Complex Refractive Index Model on page 694	1
d_n_temp (Plot name: ComplexRefractiveIndex)	vertex	Complex Refractive Index Model on page 694	1
DeepLevels	vertex	Energetic and Spatial Distribution of Traps on page 544	cm <sup>-3</sup>
DelVorWeight	vertex	Weighted Voronoï Diagram on page 1186	μm <sup>2</sup>
DielectricConstant	vertex, element	$\epsilon$ , Equation 37	1
DielectricConstantAniso	vertex, element	$\epsilon_{\text{aniso}}$ , Anisotropic Electrical Permittivity on page 901	1
DisplaceCurrentDensity	vertex	$ J_D $	Acm <sup>-2</sup>

## Appendix F: Data Names and Plot Names

### Scalar Data

Table 195    Scalar data (Continued)

Data name	Location	Description	Unit
DonorConcentration	vertex	Specifying Doping Species on page 60	$\text{cm}^{-3}$
DonorPlusConcentration	vertex		$\text{cm}^{-3}$
DopingConcentration	vertex		$\text{cm}^{-3}$
DopingWells	vertex	Indices of doping wells, <a href="#">Initial Guess for Electrostatic Potential and Quasi-Fermi Potentials in Doping Wells on page 234</a>	1
eAllValleyOccupation	vertex	$n_B/n$ , <a href="#">Equation 194</a> and <a href="#">Using Multivalley Band Structure on page 331</a>	1
eAlphaAvalanche	vertex	$\alpha_n$ , <a href="#">Equation 454</a>	$\text{cm}^{-1}$
eAugerRecombination	vertex	$R_n^A$ , <a href="#">Equation 441</a>	$\text{cm}^{-3} \text{s}^{-1}$
eBand2BandGeneration	vertex	Dynamic Nonlocal Path Band-to-Band Tunneling Model on page 529	$\text{cm}^{-3} \text{s}^{-1}$
eBandTailMobility	vertex	$\mu_{bt}$ , <a href="#">Equation 378</a>	$\text{cm}^2 \text{V}^{-1} \text{s}^{-1}$
eBandTailOccupation	vertex	$n_{bt}/n$ , <a href="#">Equation 194</a>	1
eBarrierTunneling	vertex	Nonlocal Tunneling at Interfaces, Contacts, and Junctions on page 826	$\text{cm}^{-3} \text{s}^{-1}$
eCDL1Lifetime	vertex	$\tau_{n1}$ , <a href="#">Coupled Defect Level Recombination on page 487</a>	s
eCDL2Lifetime	vertex	$\tau_{n2}$ , <a href="#">Coupled Defect Level Recombination on page 487</a>	s

## Appendix F: Data Names and Plot Names

### Scalar Data

Table 195    Scalar data (Continued)

Data name	Location	Description	Unit
eCNTMobility	vertex	$\mu$ , <a href="#">Equation 350</a>	$\text{cm}^2 \text{V}^{-1} \text{s}^{-1}$
eColHeat	vertex	$dW_n/dt _{\text{coll}}$ , <a href="#">Equation 76</a>	$\text{Wcm}^{-3}$
eCoulomb2DMobility	element	<a href="#">Chapter 15 on page 385</a>	$\text{cm}^2 \text{V}^{-1} \text{s}^{-1}$
eCoulomb3DMobility	element	<a href="#">Chapter 15 on page 385</a>	$\text{cm}^2 \text{V}^{-1} \text{s}^{-1}$
eCoulombMobility	element	<a href="#">Chapter 15 on page 385</a>	$\text{cm}^2 \text{V}^{-1} \text{s}^{-1}$
eCurrentDensity	vertex	$ J_n $ , <a href="#">Equation 57</a>	$\text{Acm}^{-2}$
eDeformationPotential	vertex	<a href="#">Using Deformation Potential Model on page 943</a>	eV
eDensity	vertex	$n$ , <a href="#">Equation 57</a>	$\text{cm}^{-3}$
eDifferentialGain	vertex	<a href="#">Stimulated and Spontaneous Emission Coefficients on page 1077</a>	$\text{cm}^2$
eDiffusivityMobility	vertex	$\mu_{n,\text{diff}}$ <a href="#">Non-Einstein Diffusivity on page 453</a>	$\text{cm}^2 \text{V}^{-1} \text{s}^{-1}$
eDopingDependentMobility	element	<a href="#">Chapter 15 on page 385</a>	$\text{cm}^2 \text{V}^{-1} \text{s}^{-1}$
eDriftVelocity	vertex	Electron drift velocity	$\text{cm s}^{-1}$
eeDiffusionLNS	vertex	<a href="#">Table 142 on page 816</a>	$\text{C}^2 \text{s}^{-1} \text{cm}^{-1}$
eEffectiveField	vertex	$E_n^{\text{eff}}$ , <a href="#">Equation 482</a>	$\text{Vcm}^{-1}$
eEffectiveStateDensity	vertex	$N_C$ , <a href="#">Effective Masses and Effective Density-of-States on page 319</a>	$\text{cm}^{-3}$

## Appendix F: Data Names and Plot Names

### Scalar Data

Table 195    Scalar data (Continued)

Data name	Location	Description	Unit
eEffectiveStress	vertex	Effective Stress on page 991	MPa
eeFlickerGRLNS	vertex	Table 142 on page 816	C <sup>2</sup> s <sup>-1</sup> cm <sup>-1</sup>
eeMonopolarGRLNS	vertex	Table 142 on page 816	C <sup>2</sup> s <sup>-1</sup> cm <sup>-1</sup>
eEnormal	vertex	$F_{\perp}$ , Equation 336 or $F_{n,\perp}$ , Equation 337	Vcm <sup>-1</sup>
eEparallel	vertex	$F_n$ , Equation 363	Vcm <sup>-1</sup>
eEquilibriumDensity	vertex	$n$ , Equation 57 at zero applied voltages (zero currents)	cm <sup>-3</sup>
EffectiveBandGap	vertex	$E_g - E_{bgn}$ , Bandgap and Electron-Affinity Models on page 305	eV
EffectiveIntrinsicDensity	vertex	$n_{i,eff}$ , Equation 152	cm <sup>-3</sup>
eGapStatesRecombination	vertex	Chapter 17 on page 543	cm <sup>-3</sup> s <sup>-1</sup>
eGradQuasiFermi	vertex	$ \nabla\Phi_n $ , Equation 365	Vcm <sup>-1</sup>
eHeatFlux	vertex	$ S_n $ , Equation 79	Wcm <sup>-2</sup>
eImpactIonization	vertex	$G_n$ , Equation 454	cm <sup>-3</sup> s <sup>-1</sup>
eInterfaceTrapConcentration	vertex	Chapter 17 on page 543	cm <sup>-2</sup>
eInterfaceTrappedCharge	vertex	Chapter 17 on page 543	cm <sup>-2</sup>
eIonIntegral	vertex	Approximate Breakdown Analysis on page 511	1

## Appendix F: Data Names and Plot Names

### Scalar Data

Table 195    *Scalar data (Continued)*

Data name	Location	Description	Unit
eJEheat	vertex	$\vec{J}_n \cdot \nabla E_C / q$ , <a href="#">Equation 76</a>	$\text{Wcm}^{-3}$
eJouleHeat	vertex	<a href="#">Table 28 on page 251</a>	$\text{Wcm}^{-3}$
ElectricField	vertex, element	$F$	$\text{Vcm}^{-1}$
ElectronAffinity	vertex	$\chi$ , <a href="#">Bandgap and Electron-Affinity Models on page 305</a>	eV
ElectrostaticPotential	vertex	$\phi$ , <a href="#">Equation 37</a>	V
eLifetime	vertex	$\tau_n$ , <a href="#">Equation 397</a>	s
eMLDAQuantumPotential	vertex	<a href="#">Modified Local-Density Approximation Model on page 370</a>	eV
eMobility	vertex, element	$\mu_n$ , <a href="#">Chapter 15 on page 385</a>	$\text{cm}^2 \text{V}^{-1} \text{s}^{-1}$
eMobilityAniso	vertex, element	$\mu_n^{\text{aniso}}$ , <a href="#">Anisotropic Mobility on page 893</a>	$\text{cm}^2 \text{V}^{-1} \text{s}^{-1}$
eMobilityAnisoFactor	vertex	$r_e$ , <a href="#">Equation 921</a>	1
eMobilityStressFactorXX	vertex	<a href="#">Returns a component of the mobility stress factor tensor, see <a href="#">Using Piezoresistance Mobility Model on page 979</a></a>	1
eMobilityStressFactorXY	vertex		1
eMobilityStressFactorXZ	vertex		1
eMobilityStressFactorYY	vertex		1
eMobilityStressFactorYZ	vertex		1
eMobilityStressFactorZZ	vertex		1

## Appendix F: Data Names and Plot Names

### Scalar Data

Table 195    *Scalar data (Continued)*

Data name	Location	Description	Unit
eMVQMBandgapShift	vertex	Modified Local-Density Approximation Model on page 370	eV
eNLLTunnelingPeltierHeat	vertex	Nonlocal Tunneling at Interfaces, Contacts, and Junctions on page 826	Wcm <sup>-3</sup>
ePhonon2DMobility	element	<a href="#">Chapter 15 on page 385</a>	cm <sup>2</sup> V <sup>-1</sup> s <sup>-1</sup>
ePhonon3DMobility	element	<a href="#">Chapter 15 on page 385</a>	cm <sup>2</sup> V <sup>-1</sup> s <sup>-1</sup>
ePhononMobility	element	<a href="#">Chapter 15 on page 385</a>	cm <sup>2</sup> V <sup>-1</sup> s <sup>-1</sup>
eQDDGamma	vertex	$\gamma$ for electrons, <a href="#">Density Gradient Model on page 362</a>	1
eQMDensity	vertex	<a href="#">Notes on the Use of the Density Gradient Model on page 367</a>	cm <sup>-3</sup>
eQuantumPotential	vertex	$\Lambda_n$ , <a href="#">Equation 224</a>	eV
eQuasiFermiEnergy	vertex	$E_{F,n}$ , <a href="#">Quasi-Fermi Potential With Boltzmann Statistics on page 231</a>	eV
eQuasiFermiPotential	vertex	$\Phi_n$ , <a href="#">Quasi-Fermi Potential With Boltzmann Statistics on page 231</a>	V
EquilibriumPotential	vertex	$\phi$ , <a href="#">Equation 37</a> at zero applied voltages (zero currents)	V
eRecGenHeat	vertex	$H_n$ , <a href="#">Equation 90</a>	Wcm <sup>-3</sup>

## Appendix F: Data Names and Plot Names

### Scalar Data

Table 195    Scalar data (Continued)

Data name	Location	Description	Unit
eRelativeEffectiveMass	vertex	$m_n$ , Effective Masses and Effective Density-of-States on page 319	1
eSaturationVelocity	vertex	$v_{\text{sat},n}$ , Velocity Saturation Models on page 446	$\text{cm s}^{-1}$
eSaturationVelocityAniso	vertex	$v_{\text{sat},n}^{\text{aniso}}$ , Anisotropic Mobility on page 893	$\text{cm s}^{-1}$
eSchenkBGN	vertex	$-\Lambda_n$ , Equation 224	eV
eSchenkTunnel	vertex	Direct Tunneling on page 822	$\text{Acm}^{-2}$
eSHEAvalancheGeneration	vertex	$G_{n, \text{SHE}}^{\text{ii}}$ , Spherical Harmonics Expansion Method on page 853	$\text{cm}^{-3} \text{s}^{-1}$
eSHECurrentDensity	vertex	$ J_{n, \text{SHE}} $ , Spherical Harmonics Expansion Method on page 853	$\text{Acm}^{-2}$
eSHEDensity	vertex	$n_{\text{SHE}}$ , Spherical Harmonics Expansion Method on page 853	$\text{cm}^{-3}$
eSHEEnergy	vertex	$T_{n, \text{SHE}}$ , Spherical Harmonics Expansion Method on page 853	K
eSHEVelocity	vertex	$ v_{n, \text{SHE}} $ , Spherical Harmonics Expansion Method on page 853	$\text{cm s}^{-1}$
eSRHRecombination	vertex	Dynamic Nonlocal Path Trap-Assisted Tunneling on page 481	$\text{cm}^{-3} \text{s}^{-1}$

## Appendix F: Data Names and Plot Names

### Scalar Data

Table 195    Scalar data (Continued)

Data name	Location	Description	Unit
eSurfRoughnessMobility	element	<a href="#">Chapter 15 on page 385</a>	$\text{cm}^2 \text{V}^{-1} \text{s}^{-1}$
eTBulkPhononMobility	element	<a href="#">Chapter 15 on page 385</a>	$\text{cm}^2 \text{V}^{-1} \text{s}^{-1}$
eTemperature	vertex	$T_n$ , <a href="#">Hydrodynamic Model for Temperatures on page 252</a>	K
eTemperatureRelaxationTime	vertex	$\tau_{en}$ , <a href="#">Equation 87</a>	s
eTensorMobilityFactorXX	vertex	<a href="#">Chapter 31 on page 935</a>	1
eTensorMobilityFactorYY	vertex	<a href="#">Chapter 31 on page 935</a>	1
eTensorMobilityFactorZZ	vertex	<a href="#">Chapter 31 on page 935</a>	1
eTensorMobilityXX	vertex	<a href="#">Chapter 31 on page 935</a>	$\text{cm}^2 / (\text{Vs})$
eTensorMobilityYY	vertex	<a href="#">Chapter 31 on page 935</a>	$\text{cm}^2 / (\text{Vs})$
eTensorMobilityZZ	vertex	<a href="#">Chapter 31 on page 935</a>	$\text{cm}^2 / (\text{Vs})$
eTFluctuationMobility	element	<a href="#">Chapter 15 on page 385</a>	$\text{cm}^2 \text{V}^{-1} \text{s}^{-1}$
eThermoElectricPower	vertex	$P_n$ , <a href="#">Equation 1127</a>	$\text{V K}^{-1}$
eTrapConcentration	vertex	<a href="#">Chapter 17 on page 543</a>	$\text{cm}^{-3}$
eTrappedCharge	vertex	<a href="#">Chapter 17 on page 543</a>	$\text{cm}^{-3}$
eTSurfacePhononMobility	element	<a href="#">Chapter 15 on page 385</a>	$\text{cm}^2 \text{V}^{-1} \text{s}^{-1}$
eValleyOccupation_name	vertex	$n_i/n$ , <a href="#">Using Multivalley Band Structure on page 331</a>	1
eVelocity	vertex	$v_n = \left  \vec{J}_n / nq \right $	$\text{cm s}^{-1}$

## Appendix F: Data Names and Plot Names

### Scalar Data

Table 195 Scalar data (Continued)

Data name	Location	Description	Unit
f1BandOccupancy001	vertex	<a href="#">Using Intel Mobility Model on page 975</a>	1
f1BandOccupancy010	vertex		1
f1BandOccupancy100	vertex		1
f2BandOccupancy001	vertex		1
f2BandOccupancy010	vertex		1
f2BandOccupancy100	vertex		1
FowlerNordheim	vertex	$j_{FN}$ , <a href="#">Equation 806</a>	$\text{A cm}^{-2}$
Grad2PoECACGreenFunction	vertex	<a href="#">Table 142 on page 816</a>	$\text{V}^2 \text{s}^2 \text{C}^{-2} \text{cm}^{-2}$
Grad2PoHCACGreenFunction	vertex	<a href="#">Table 142 on page 816</a>	$\text{V}^2 \text{s}^2 \text{C}^{-2} \text{cm}^{-2}$
hAllValleyOccupation	vertex	$n_B/n$ , <a href="#">Equation 194</a> and <a href="#">Using Multivalley Band Structure on page 331</a>	1
hAlphaAvalanche	vertex	$\alpha_p$ , <a href="#">Equation 454</a>	$\text{cm}^{-1}$
hAugerRecombination	vertex	$R_p^A$ , <a href="#">Equation 441</a>	$\text{cm}^{-3} \text{s}^{-1}$
hBand2BandGeneration	vertex	Dynamic Nonlocal Path Band-to-Band Tunneling Model on page 529	$\text{cm}^{-3} \text{s}^{-1}$
hBandTailMobility	vertex	$\mu_{bt}$ , <a href="#">Equation 378</a>	$\text{cm}^2 \text{V}^{-1} \text{s}^{-1}$
hBandTailOccupation	vertex	$n_{bt}/n$ , <a href="#">Equation 194</a>	1
hBarrierTunneling	vertex	Nonlocal Tunneling at Interfaces, Contacts, and Junctions on page 826	$\text{cm}^{-3} \text{s}^{-1}$

## Appendix F: Data Names and Plot Names

### Scalar Data

Table 195    *Scalar data (Continued)*

Data name	Location	Description	Unit
hCDL1Lifetime	vertex	$\tau_{p1}$ , <a href="#">Coupled Defect Level Recombination on page 487</a>	s
hCDL2Lifetime	vertex	$\tau_{p2}$ , <a href="#">Coupled Defect Level Recombination on page 487</a>	s
hCNTMobility	vertex	$\mu$ , <a href="#">Equation 350</a>	$\text{cm}^2 \text{V}^{-1} \text{s}^{-1}$
hColHeat	vertex	$dW_p/dt _{\text{coll}}$ , <a href="#">Equation 77</a>	$\text{Wcm}^{-3}$
hCoulomb2DMobility	element	<a href="#">Chapter 15 on page 385</a>	$\text{cm}^2 \text{V}^{-1} \text{s}^{-1}$
hCoulomb3DMobility	element	<a href="#">Chapter 15 on page 385</a>	$\text{cm}^2 \text{V}^{-1} \text{s}^{-1}$
hCoulombMobility	element	<a href="#">Chapter 15 on page 385</a>	$\text{cm}^2 \text{V}^{-1} \text{s}^{-1}$
hCurrentDensity	vertex	$ J_p $ , <a href="#">Equation 57</a>	$\text{Acm}^{-2}$
hDeformationPotential	vertex	<a href="#">Using Deformation Potential Model on page 943</a>	eV
hDensity	vertex	$p$ , <a href="#">Equation 57</a>	$\text{cm}^{-3}$
hDifferentialGain	vertex	<a href="#">Stimulated and Spontaneous Emission Coefficients on page 1077</a>	$\text{cm}^2$
hDiffusivityMobility	vertex	$\mu_{p, \text{diff}}$ <a href="#">Non-Einstein Diffusivity on page 453</a>	$\text{cm}^2 \text{V}^{-1} \text{s}^{-1}$
hDopingDependentMobility	element	<a href="#">Chapter 15 on page 385</a>	$\text{cm}^2 \text{V}^{-1} \text{s}^{-1}$
hDriftVelocity	vertex	Hole drift velocity	$\text{cm s}^{-1}$

## Appendix F: Data Names and Plot Names

### Scalar Data

Table 195    Scalar data (Continued)

Data name	Location	Description	Unit
HeavyIonChargeDensity	vertex	Carrier Generation by Heavy Ions on page 775	$\text{cm}^{-3}$
HeavyIonGeneration	vertex	$G^{\text{HeavyIon}}$ , Equation 782	$\text{cm}^{-3} \text{s}^{-1}$
hEffectiveField	vertex	$E_p^{\text{eff}}$ , Equation 483	$\text{V cm}^{-1}$
hEffectiveStateDensity	vertex	$N_V$ , Effective Masses and Effective Density-of-States on page 319	$\text{cm}^{-3}$
hEffectiveStress	vertex	Effective Stress on page 991	MPa
heiTemperature	vertex	$T_{\text{hei}}$ , hot-electron temperature, computed as postprocessing approach (carrier TempPost), Chapter 25 on page 844	K
hEnormal	vertex	$F_{\perp}$ , Equation 336 or $F_{p,\perp}$ , Equation 337	$\text{V cm}^{-1}$
hEparallel	vertex	$F_p$ , Equation 363	$\text{V cm}^{-1}$
hEquilibriumDensity	vertex	$p$ , Equation 57 at zero applied voltages (zero currents)	$\text{cm}^{-3}$
hGapStatesRecombination	vertex	Chapter 17 on page 543	$\text{cm}^{-3} \text{s}^{-1}$
hGradQuasiFermi	vertex	$ \nabla \Phi_p $ , Equation 365	$\text{V cm}^{-1}$
hhDiffusionLNS	vertex	Table 142 on page 816	$\text{C}^2 \text{s}^{-1} \text{cm}^{-1}$
hHeatFlux	vertex	$ S_p $ , Equation 80	$\text{W cm}^{-2}$

## Appendix F: Data Names and Plot Names

### Scalar Data

*Table 195 Scalar data (Continued)*

Data name	Location	Description	Unit
hhFlickerGRLNS	vertex	<a href="#">Table 142 on page 816</a>	$C^2 s^{-1} cm^{-1}$
hhMonopolarGRLNS	vertex	<a href="#">Table 142 on page 816</a>	$C^2 s^{-1} cm^{-1}$
HighFieldEntranceMask	vertex, element	<a href="#">High-Field Entrance Position and Time on page 520</a>	1
HighFieldEntranceTime	vertex, element	<a href="#">High-Field Entrance Position and Time on page 520</a>	1
HighFieldEntranceVertex	element	<a href="#">High-Field Entrance Position and Time on page 520</a>	1
hImpactIonization	vertex	$G_p$ , <a href="#">Equation 454</a>	$cm^{-3} s^{-1}$
hInterfaceTrapConcentration	vertex	<a href="#">Chapter 17 on page 543</a>	$cm^{-2}$
hInterfaceTrappedCharge	vertex	<a href="#">Chapter 17 on page 543</a>	$cm^{-2}$
hIonIntegral	vertex	<a href="#">Approximate Breakdown Analysis on page 511</a>	1
hJEHeat	vertex	$\vec{j}_p \cdot \nabla E_V / q$ , <a href="#">Equation 77</a>	$W cm^{-3}$
hJouleHeat	vertex	<a href="#">Table 28 on page 251</a>	$W cm^{-3}$
hLifetime	vertex	$\tau_p$ , <a href="#">Equation 397</a>	s
hMLDAQuantumPotential	vertex	<a href="#">Modified Local-Density Approximation Model on page 370</a>	eV
hMobility	vertex, element	$\mu_p$ , <a href="#">Chapter 15 on page 385</a>	$cm^2 V^{-1} s^{-1}$
hMobilityAniso	vertex, element	$\mu_p^{aniso}$ , <a href="#">Anisotropic Mobility on page 893</a>	$cm^2 V^{-1} s^{-1}$

## Appendix F: Data Names and Plot Names

### Scalar Data

Table 195    Scalar data (Continued)

Data name	Location	Description	Unit
hMobilityAnisoFactor	vertex	$r_h$ , <a href="#">Equation 921</a>	1
hMobilityStressFactorXX	vertex	<a href="#">Using Piezoresistance Mobility Model on page 979</a>	1
hMobilityStressFactorXY	vertex		1
hMobilityStressFactorXZ	vertex		1
hMobilityStressFactorYY	vertex		1
hMobilityStressFactorYZ	vertex		1
hMobilityStressFactorZZ	vertex		1
hMVQMBandgapShift	vertex	<a href="#">Modified Local-Density Approximation Model on page 370</a>	eV
hNLLTunnelingPeltierHeat	vertex	<a href="#">Nonlocal Tunneling at Interfaces, Contacts, and Junctions on page 826</a>	Wcm <sup>-3</sup>
HotElectronInj	vertex	Hot-electron current density $j_{he}$ at interface, <a href="#">Equation 841</a> , <a href="#">Equation 847</a>	Acm <sup>-2</sup>
HotHoleInj	vertex	Hot-hole current density $j_{hh}$ at interface, <a href="#">Equation 841</a> , <a href="#">Equation 847</a>	Acm <sup>-2</sup>
hPhonon2DMobility	element	<a href="#">Chapter 15 on page 385</a>	cm <sup>2</sup> V <sup>-1</sup> s <sup>-1</sup>
hPhonon3DMobility	element	<a href="#">Chapter 15 on page 385</a>	cm <sup>2</sup> V <sup>-1</sup> s <sup>-1</sup>
hPhononMobility	element	<a href="#">Chapter 15 on page 385</a>	cm <sup>2</sup> V <sup>-1</sup> s <sup>-1</sup>
hQDDGamma	vertex	$\gamma$ for holes, <a href="#">Density Gradient Model on page 362</a>	1

## Appendix F: Data Names and Plot Names

### Scalar Data

Table 195    Scalar data (Continued)

Data name	Location	Description	Unit
hQMDensity	vertex	Notes on the Use of the Density Gradient Model on page 367	cm <sup>-3</sup>
hQuantumPotential	vertex	$\Lambda_p$ , Equation 224	eV
hQuasiFermiEnergy	vertex	$E_{F,p}$ , Quasi-Fermi Potential With Boltzmann Statistics on page 231	eV
hQuasiFermiPotential	vertex	$\Phi_p$ , Quasi-Fermi Potential With Boltzmann Statistics on page 231	V
hRecGenHeat	vertex	$H_p$ , Equation 91	Wcm <sup>-3</sup>
hRelativeEffectiveMass	vertex	$m_p$ , Effective Masses and Effective Density-of-States on page 319	1
hSaturationVelocity	vertex	$v_{\text{sat},p}$ , Velocity Saturation Models on page 446	cm s <sup>-1</sup>
hSaturationVelocityAniso	vertex	$v_{\text{sat},p}^{\text{aniso}}$ , Anisotropic Mobility on page 893	cm s <sup>-1</sup>
hSchenkBGN	vertex	$-\Lambda_p$ , Equation 224	eV
hSchenkTunnel	vertex	Direct Tunneling on page 822	Acm <sup>-2</sup>
hSHEAvalancheGeneration	vertex	$G_{p,\text{SHE}}^{\text{ii}}$ , Spherical Harmonics Expansion Method on page 853	cm <sup>-3</sup> s <sup>-1</sup>
hSHECurrentDensity	vertex	$ J_{p,\text{SHE}} $ , Spherical Harmonics Expansion Method on page 853	Acm <sup>-2</sup>

## Appendix F: Data Names and Plot Names

### Scalar Data

*Table 195 Scalar data (Continued)*

Data name	Location	Description	Unit
hSHEDensity	vertex	$p_{\text{SHE}}$ , Spherical Harmonics Expansion Method on page 853	$\text{cm}^{-3}$
hSHEEnergy	vertex	$T_{p, \text{SHE}}$ , Spherical Harmonics Expansion Method on page 853	K
hSHEVelocity	vertex	$ v_{p, \text{SHE}} $ , Spherical Harmonics Expansion Method on page 853	$\text{cm s}^{-1}$
hSRHRecombination	vertex	Dynamic Nonlocal Path Trap-Assisted Tunneling on page 481	$\text{cm}^{-3} \text{s}^{-1}$
hSurfRoughnessMobility	element	<a href="#">Chapter 15 on page 385</a>	$\text{cm}^2 \text{V}^{-1} \text{s}^{-1}$
hTBulkPhononMobility	element	<a href="#">Chapter 15 on page 385</a>	$\text{cm}^2 \text{V}^{-1} \text{s}^{-1}$
hTemperature	vertex	$T_p$ , Hydrodynamic Model for Temperatures on page 252	K
hTemperatureRelaxationTime	vertex	$\tau_{ep}$ , Equation 88	s
hTensorMobilityFactorXX	vertex	<a href="#">Chapter 31 on page 935</a>	1
hTensorMobilityFactorYY	vertex	<a href="#">Chapter 31 on page 935</a>	1
hTensorMobilityFactorZZ	vertex	<a href="#">Chapter 31 on page 935</a>	1
hTensorMobilityXX	vertex	<a href="#">Chapter 31 on page 935</a>	$\text{cm}^2 / (\text{Vs})$
hTensorMobilityYY	vertex	<a href="#">Chapter 31 on page 935</a>	$\text{cm}^2 / (\text{Vs})$
hTensorMobilityZZ	vertex	<a href="#">Chapter 31 on page 935</a>	$\text{cm}^2 / (\text{Vs})$
hTFluctuationMobility	element	<a href="#">Chapter 15 on page 385</a>	$\text{cm}^2 \text{V}^{-1} \text{s}^{-1}$

## Appendix F: Data Names and Plot Names

### Scalar Data

Table 195 Scalar data (Continued)

Data name	Location	Description	Unit
hThermoElectricPower	vertex	$P_p$ , <a href="#">Equation 1128</a>	V K <sup>-1</sup>
hTrapConcentration	vertex	<a href="#">Chapter 17 on page 543</a>	cm <sup>-3</sup>
hTrappedCharge	vertex	<a href="#">Chapter 17 on page 543</a>	cm <sup>-3</sup>
hTSurfacePhononMobility	element	<a href="#">Chapter 15 on page 385</a>	cm <sup>2</sup> V <sup>-1</sup> s <sup>-1</sup>
hValleyOccupation_name	vertex	$n_i/n$ , <a href="#">Using Multivalley Band Structure on page 331</a>	1
hVelocity	vertex	$v_p = \left  \vec{J}_p / pq \right $	cm s <sup>-1</sup>
HydrogenAtom	vertex	<a href="#">MSC-Hydrogen Transport Degradation Model on page 611</a>	cm <sup>-3</sup>
HydrogenIon	vertex		cm <sup>-3</sup>
HydrogenMolecule	vertex		cm <sup>-3</sup>
HydrogenSpeciesA	vertex		cm <sup>-3</sup>
HydrogenSpeciesB	vertex		cm <sup>-3</sup>
HydrogenSpeciesC	vertex		cm <sup>-3</sup>
ImeDensityResponse	vertex	$\text{Im}(\tilde{n})$ <a href="#">AC Response on page 1192</a>	cm <sup>-3</sup> V <sup>-1</sup>
ImeeDiffusionLNVXVSD	vertex	<a href="#">Table 142 on page 816</a>	V <sup>2</sup> scm <sup>-3</sup>
ImeeFlickerGRLNVXVSD	vertex	<a href="#">Table 142 on page 816</a>	V <sup>2</sup> scm <sup>-3</sup>
ImeeLNVXVSD	vertex	<a href="#">Table 142 on page 816</a>	V <sup>2</sup> scm <sup>-3</sup>
ImeeMonopolarGRLNVXVSD	vertex	<a href="#">Table 142 on page 816</a>	V <sup>2</sup> scm <sup>-3</sup>

## Appendix F: Data Names and Plot Names

### Scalar Data

Table 195    Scalar data (Continued)

Data name	Location	Description	Unit
ImElectrostaticPotentialResponse	vertex	Im( $\tilde{\phi}$ ) <a href="#">AC Response on page 1192</a>	1
ImeTemperatureResponse	vertex	Im( $\tilde{T}_n$ ) <a href="#">AC Response on page 1192</a>	KV <sup>-1</sup>
ImhDensityResponse	vertex	Im( $\tilde{p}$ ) <a href="#">AC Response on page 1192</a>	cm <sup>-3</sup> V <sup>-1</sup>
ImhhDiffusionLNVXVSD	vertex	<a href="#">Table 142 on page 816</a>	V <sup>2</sup> scm <sup>-3</sup>
ImhhFlickerGRLNVXVSD	vertex	<a href="#">Table 142 on page 816</a>	V <sup>2</sup> scm <sup>-3</sup>
ImhhLNVXVSD	vertex	<a href="#">Table 142 on page 816</a>	V <sup>2</sup> scm <sup>-3</sup>
ImhhMonopolarGRLNVXVSD	vertex	<a href="#">Table 142 on page 816</a>	V <sup>2</sup> scm <sup>-3</sup>
ImhTemperatureResponse	vertex	Im( $\tilde{T}_p$ ) <a href="#">AC Response on page 1192</a>	KV <sup>-1</sup>
ImLatticeTemperatureResponse	vertex	Im( $\tilde{T}$ ) <a href="#">AC Response on page 1192</a>	KV <sup>-1</sup>
ImLNISD	vertex	<a href="#">Table 142 on page 816</a>	A <sup>2</sup> scm <sup>-3</sup>
ImLNVXVSD	vertex	<a href="#">Table 142 on page 816</a>	V <sup>2</sup> scm <sup>-3</sup>
ImpactIonization	vertex	$G^{\parallel}$ , <a href="#">Equation 454</a>	cm <sup>-3</sup> s <sup>-1</sup>
ImTrapLNISD	vertex	<a href="#">Table 142 on page 816</a>	A <sup>2</sup> scm <sup>-3</sup>
ImTrapLNVSD	vertex	<a href="#">Table 142 on page 816</a>	V <sup>2</sup> scm <sup>-3</sup>
IndiumActiveConcentration	vertex	<a href="#">Specifying Doping Species on page 60</a>	cm <sup>-3</sup>
IndiumConcentration	vertex	In, <a href="#">Specifying Doping Species on page 60</a>	cm <sup>-3</sup>

## Appendix F: Data Names and Plot Names

### Scalar Data

Table 195    *Scalar data (Continued)*

Data name	Location	Description	Unit
IndiumMinusConcentration	vertex	In <sup>-</sup> , <a href="#">Chapter 13 on page 339</a>	cm <sup>-3</sup>
InsulatorElectricField	vertex	Electric field $F$ on insulator	V cm <sup>-1</sup>
InterfaceNBTICharge	vertex	<a href="#">Two-Stage NBTI Degradation Model on page 621</a>	cm <sup>-2</sup>
InterfaceNBTIState1	vertex		cm <sup>-2</sup>
InterfaceNBTIState2	vertex		cm <sup>-2</sup>
InterfaceNBTIState3	vertex		cm <sup>-2</sup>
InterfaceNBTIState4	vertex		cm <sup>-2</sup>
InterfaceOrientation	vertex	<a href="#">Auto-Orientation Framework on page 86</a>	1
IntrinsicDensity	vertex	$n_i$ , <a href="#">Equation 151</a>	cm <sup>-3</sup>
IonizedDopingConcentration	vertex	<a href="#">Specifying Doping Species on page 60, Using the Incomplete Ionization Model on page 339</a>	cm <sup>-3</sup>
JouleHeat	vertex	<a href="#">Table 28 on page 251</a>	W cm <sup>-3</sup>
k_0 (Plot name: ComplexRefractiveIndex)	vertex	<a href="#">Complex Refractive Index Model on page 694</a>	1
LatticeHeatCapacity	vertex	$c_L$ , <a href="#">Heat Capacity on page 1017</a>	JK <sup>-1</sup> cm <sup>-3</sup>
LatticeTemperature	vertex	$T$ , <a href="#">Chapter 9 on page 245</a>	K
LayerThickness	vertex	<a href="#">Extracting Layer Thickness on page 379</a>	μm

## Appendix F: Data Names and Plot Names

### Scalar Data

*Table 195 Scalar data (Continued)*

Data name	Location	Description	Unit
LayerThicknessField	vertex	<a href="#">Extracting Layer Thickness on page 379</a>	μm
lColHeat	vertex	$dW_L/dt _{\text{coll}}$ , <a href="#">Equation 78</a>	Wcm <sup>-3</sup>
lHeatFlux	vertex	$  \vec{S}_L  $ , <a href="#">Equation 81</a>	Wcm <sup>-2</sup>
lRecGenHeat	vertex	$H_L$ , <a href="#">Equation 92</a>	Wcm <sup>-3</sup>
MeanIonIntegral	vertex	<a href="#">Approximate Breakdown Analysis on page 511</a>	1
MetalConductivity	vertex	$\sigma$ , <a href="#">Transport in Metals on page 295</a>	Acm <sup>-1</sup> V <sup>-1</sup>
MetalWorkfunction	vertex	<a href="#">Metal Workfunction on page 298</a>	eV
MIMCellPotential	vertex	<a href="#">Concentration Plots on page 1140</a>	V
MIMCellTemperature	vertex		K
MIMChargeConc	vertex		cm <sup>-3</sup>
MIMDefect1Conc	vertex		cm <sup>-3</sup>
MIMDefect2Conc	vertex		cm <sup>-3</sup>
MIMDefectConc	vertex		cm <sup>-3</sup>
MIMElectronConc	vertex		cm <sup>-3</sup>
MobilityAcceptorConcentration	vertex	<a href="#">Mobility Doping File on page 465</a> or <code>Add2TotalDoping(ChargedTraps), Specifying Doping Species on page 60</code>	cm <sup>-3</sup>

## Appendix F: Data Names and Plot Names

### Scalar Data

*Table 195 Scalar data (Continued)*

Data name	Location	Description	Unit
MobilityDonorConcentration	vertex	<a href="#">Mobility Doping File on page 465 or Add2TotalDoping (ChargedTraps), Specifying Doping Species on page 60</a>	cm <sup>-3</sup>
Mod_eGradQuasiFermi_ElectricField	vertex	$ \tilde{\nabla\Phi_n} $ , <a href="#">Interpolation of Driving Forces to Zero Field on page 450</a>	Vcm <sup>-1</sup>
Mod_eQuasiFermi_ElectricField_Potential	vertex	$\tilde{\Phi_n}$ , <a href="#">Interpolation of Driving Forces to Zero Field on page 450</a>	V
Mod_hGradQuasiFermi_ElectricField	vertex	$ \tilde{\nabla\Phi_p} $ , <a href="#">Interpolation of Driving Forces to Zero Field on page 450</a>	Vcm <sup>-1</sup>
Mod_hQuasiFermi_ElectricField_Potential	vertex	$\tilde{\Phi_p}$ , <a href="#">Interpolation of Driving Forces to Zero Field on page 450</a>	V
n_0 (Plot name: ComplexRefractiveIndex)	vertex	<a href="#">Complex Refractive Index Model on page 694</a>	1
NDopantActiveConcentration	vertex	<a href="#">Specifying Doping Species on page 60</a>	cm <sup>-3</sup>
NDopantConcentration	vertex	NDopant, <a href="#">Specifying Doping Species on page 60</a>	cm <sup>-3</sup>
NDopantPlusConcentration	vertex	NDopant+, <a href="#">Chapter 13 on page 339</a>	cm <sup>-3</sup>
NearestInterfaceOrientation	vertex	<a href="#">Auto-Orientation Framework on page 86</a>	1
NegInterfaceCharge	vertex	<a href="#">Mobility Degradation Components due to Coulomb Scattering on page 419</a>	cm <sup>-2</sup>

## Appendix F: Data Names and Plot Names

### Scalar Data

*Table 195 Scalar data (Continued)*

Data name	Location	Description	Unit
NitrogenActiveConcentration	vertex	<a href="#">Specifying Doping Species on page 60</a>	cm <sup>-3</sup>
NitrogenConcentration	vertex	<a href="#">N, Specifying Doping Species on page 60</a>	cm <sup>-3</sup>
NitrogenPlusConcentration	vertex	<a href="#">N+, Chapter 13 on page 339</a>	cm <sup>-3</sup>
OneOverDegradationTime	vertex	<a href="#">Chapter 19 on page 600</a>	s <sup>-1</sup>
OpticalAbsorptionHeat	vertex	<a href="#">Optical Absorption Heat on page 660</a>	Wcm <sup>-3</sup>
OpticalAbsorptionHeat_Bandgap (Plot name: OpticalAbsorptionHeat)	vertex	<a href="#">Optical Absorption Heat on page 660</a>	Wcm <sup>-3</sup>
OpticalAbsorptionHeat_Vacuum (Plot name: OpticalAbsorptionHeat)	vertex	<a href="#">Optical Absorption Heat on page 660</a>	Wcm <sup>-3</sup>
OpticalField	vertex	Plot optical intensity as well as real and imaginary parts of optical field	W/m <sup>-3</sup> V/m
OpticalGeneration	vertex	$G_0^{\text{opt}}$ , <a href="#">Equation 753</a>	cm <sup>-3</sup> s <sup>-1</sup>
OpticalGenerationFromConstant (Plot name: OpticalGeneration)	vertex	<a href="#">Specifying the Type of Optical Generation Computation on page 649</a>	cm <sup>-3</sup> s <sup>-1</sup>
OpticalGenerationFromFile (Plot name: OpticalGeneration)	vertex	<a href="#">Specifying the Type of Optical Generation Computation on page 649</a>	cm <sup>-3</sup> s <sup>-1</sup>
OpticalGenerationFromMonochromaticSource (Plot name: OpticalGeneration)	vertex	<a href="#">Specifying the Type of Optical Generation Computation on page 649</a>	cm <sup>-3</sup> s <sup>-1</sup>

## Appendix F: Data Names and Plot Names

### Scalar Data

*Table 195 Scalar data (Continued)*

Data name	Location	Description	Unit
OpticalGenerationFromSpectrum (Plot name: OpticalGeneration)	vertex	Specifying the Type of Optical Generation Computation on page 649	$\text{cm}^{-3} \text{s}^{-1}$
OpticalIntensity	vertex	Solving the Optical Problem on page 668	$\text{Wcm}^{-2}$
OpticalIntensityCoherent (Plot name: OpticalIntensity)	vertex	Transfer Matrix Method on page 734	$\text{Wcm}^{-2}$
OpticalIntensityIncoherent (Plot name: OpticalIntensity)	vertex	Transfer Matrix Method on page 734	$\text{Wcm}^{-2}$
ParallelToInterfaceInBoundaryLayerActive	element	Field Correction Close to Interfaces on page 452	1
PDopantActiveConcentration	vertex	Specifying Doping Species on page 60	$\text{cm}^{-3}$
PDopantConcentration	vertex	PDopant, Specifying Doping Species on page 60	$\text{cm}^{-3}$
PDopantMinusConcentration	vertex	PDopant-, Chapter 13 on page 339	$\text{cm}^{-3}$
PE_Charge	vertex	$q_{\text{PE}}$ , Piezoelectric Datasets on page 1006	$\text{cm}^{-3}$
PeltierHeat	vertex	Table 28 on page 251	$\text{Wcm}^{-3}$
PhosphorusActiveConcentration	vertex	Specifying Doping Species on page 60	$\text{cm}^{-3}$
PhosphorusConcentration	vertex	P, Specifying Doping Species on page 60	$\text{cm}^{-3}$
PhosphorusPlusConcentration	vertex	P+, Chapter 13 on page 339	$\text{cm}^{-3}$
PiezoCharge	vertex	$q_{\text{PE}}$ , Piezoelectric Datasets on page 1006	$\text{cm}^{-3}$

## Appendix F: Data Names and Plot Names

### Scalar Data

*Table 195 Scalar data (Continued)*

Data name	Location	Description	Unit
PMIENonLocalRecombination	vertex	$R_n^{\text{PMI}}$ , Nonlocal Generation–Recombination on page 1273	$\text{cm}^{-3} \text{s}^{-1}$
PMIHeat	vertex	Heat Generation Rate on page 1342	$\text{Wcm}^{-3}$
PMIHNonLocalRecombination	vertex	$R_p^{\text{PMI}}$ , Nonlocal Generation–Recombination on page 1273	$\text{cm}^{-3} \text{s}^{-1}$
PMIRecombination	vertex	$R^{\text{PMI}}$ , Generation–Recombination on page 1266	$\text{cm}^{-3} \text{s}^{-1}$
PMIUserField0	vertex	Command File of Sentaurus Device on page 1230	1
PMIUserField1	vertex		1
...	vertex		1
PMIUserField299	vertex		1
PoECImACGreenFunction	vertex	<a href="#">Table 142 on page 816</a>	$\text{VsC}^{-1}$
PoECReACGreenFunction	vertex	<a href="#">Table 142 on page 816</a>	$\text{VsC}^{-1}$
PoETImACGreenFunction	vertex	<a href="#">Table 142 on page 816</a>	$\text{A}^{-1}$
PoETReACGreenFunction	vertex	<a href="#">Table 142 on page 816</a>	$\text{A}^{-1}$
PoGeoGreenFunction	vertex	<a href="#">Table 142 on page 816</a>	$\text{Vcm}^{-3}$
PoHCImACGreenFunction	vertex	<a href="#">Table 142 on page 816</a>	$\text{VsC}^{-1}$
PoHCReACGreenFunction	vertex	<a href="#">Table 142 on page 816</a>	$\text{VsC}^{-1}$

## Appendix F: Data Names and Plot Names

### Scalar Data

*Table 195 Scalar data (Continued)*

Data name	Location	Description	Unit
PoHTImACGreenFunction	vertex	<a href="#">Table 142 on page 816</a>	A <sup>-1</sup>
PoHTReACGreenFunction	vertex	<a href="#">Table 142 on page 816</a>	A <sup>-1</sup>
PoLTImACGreenFunction	vertex	<a href="#">Table 142 on page 816</a>	A <sup>-1</sup>
PoLTReACGreenFunction	vertex	<a href="#">Table 142 on page 816</a>	A <sup>-1</sup>
PoPotImACGreenFunction	vertex	<a href="#">Table 142 on page 816</a>	VC <sup>-1</sup>
PoPotReACGreenFunction	vertex	<a href="#">Table 142 on page 816</a>	VC <sup>-1</sup>
Polarization	vertex	$\vec{P}$  , <a href="#">Chapter 29 on page 905</a>	Ccm <sup>-2</sup>
PosInterfaceCharge	vertex	<a href="#">Mobility Degradation Components due to Coulomb Scattering on page 419</a>	cm <sup>-2</sup>
QCEffectiveBandGap	vertex	$E_g - E_{bgn} - q(\Lambda_n + \Lambda_n)$	eV
QCEffectiveIntrinsicDensity	vertex	$n_{i, eff}$ with corrections due to Fermi statistics and quantization effects	cm <sup>-3</sup>
QuantumYield	vertex	$\eta_G$ , <a href="#">Equation 689</a>	1
QuantumYieldEffectiveGenerationEnergy (Plot name: QuantumYield)	vertex	$E_g$ , <a href="#">Equation 691</a>	eV
QuasiFermiPotential	vertex	$\Phi$ , <a href="#">Equation 130</a>	V
QW_chEigenEnergy	vertex	Eigenenergies of the crystal-field split-hole bound states, <a href="#">Localized Quantum-Well Model on page 1092</a>	eV

## Appendix F: Data Names and Plot Names

### Scalar Data

*Table 195 Scalar data (Continued)*

Data name	Location	Description	Unit
QW_chNumberOfBoundStates	vertex	Actual number of QW bound states for crystal-field split-holes, <a href="#">Localized Quantum-Well Model on page 1092</a>	1
QW_chRelativeEffectiveMass	vertex	Relative effective mass of crystal-field split-holes, <a href="#">Localized Quantum-Well Model on page 1092</a>	1
QW_chStrainBandShift	vertex	Shift in crystal-field split-hole band due to strain effects, <a href="#">Localized Quantum-Well Model on page 1092</a>	eV
QW_eEigenEnergy	vertex	Eigenenergies of the electron bound states, <a href="#">Localized Quantum-Well Model on page 1092</a>	eV
QW_ElectricFieldProjection	vertex	Electric field in the QW, <a href="#">Localized Quantum-Well Model on page 1092</a>	V cm <sup>-1</sup>
QW_eNumberOfBoundStates	vertex	Actual number of QW bound states for electrons, <a href="#">Localized Quantum-Well Model on page 1092</a>	1
QW_eRelativeEffectiveMass	vertex	Relative effective mass of electrons, <a href="#">Localized Quantum-Well Model on page 1092</a>	1
QW_eStrainBandShift	vertex	Shift in conduction band due to strain effects, <a href="#">Localized Quantum-Well Model on page 1092</a>	eV
QW_hhEigenEnergy	vertex	Eigenenergies of the heavy-hole bound states, <a href="#">Localized Quantum-Well Model on page 1092</a>	eV

## Appendix F: Data Names and Plot Names

### Scalar Data

*Table 195 Scalar data (Continued)*

Data name	Location	Description	Unit
QW_hhNumberOfBoundStates	vertex	Actual number of QW bound states for heavy holes, <a href="#">Localized Quantum-Well Model on page 1092</a>	1
QW_hhRelativeEffectiveMass	vertex	Relative effective mass of heavy holes, <a href="#">Localized Quantum-Well Model on page 1092</a>	1
QW_hhStrainBandShift	vertex	Shift in heavy-hole band due to strain effects, <a href="#">Localized Quantum-Well Model on page 1092</a>	eV
QW_lhEigenEnergy	vertex	Eigenenergies for the light-hole bound states, <a href="#">Localized Quantum-Well Model on page 1092</a>	eV
QW_lhNumberOfBoundStates	vertex	Actual number of QW bound states for light holes, <a href="#">Localized Quantum-Well Model on page 1092</a>	1
QW_lhRelativeEffectiveMass	vertex	Relative effective mass of light holes, <a href="#">Localized Quantum-Well Model on page 1092</a>	1
QW_lhStrainBandShift	vertex	Shift in light-hole band due to strain effects, <a href="#">Localized Quantum-Well Model on page 1092</a>	eV
QW_OverlapIntegral	vertex	Overlap integrals between electron and hole wavefunctions, <a href="#">Localized Quantum-Well Model on page 1092</a>	1

## Appendix F: Data Names and Plot Names

### Scalar Data

*Table 195 Scalar data (Continued)*

Data name	Location	Description	Unit
QW_QuantizationDirection	vertex	Quantization direction of the QW, <a href="#">Localized Quantum-Well Model on page 1092</a>	1
QW_Width	vertex	Extracted width of the QW, <a href="#">Localized Quantum-Well Model on page 1092</a>	μm
QWeDensity	vertex	$n$ , <a href="#">Equation 1207</a>	cm <sup>-3</sup>
QWeQuasiFermi	vertex	$\Phi_n$ , <a href="#">Quasi-Fermi Potential With Boltzmann Statistics on page 231</a>	V
QWhDensity	vertex	$p$ , <a href="#">Equation 1208</a>	cm <sup>-3</sup>
QWhQuasiFermi	vertex	$\Phi_p$ , <a href="#">Quasi-Fermi Potential With Boltzmann Statistics on page 231</a>	V
RadiationGeneration	vertex	$G_r$ , <a href="#">Equation 766</a>	cm <sup>-3</sup> s <sup>-1</sup>
RadiativeRecombination	vertex	$R$ , <a href="#">Equation 441</a>	cm <sup>-3</sup> s <sup>-1</sup>
RandomizedDoping	vertex	<a href="#">Statistical Impedance Field Method on page 797</a>	cm <sup>-3</sup>
RayTrees	vertex	<a href="#">Using the Raytracer on page 705</a>	1
RecombinationHeat	vertex	<a href="#">Table 28 on page 251</a>	Wcm <sup>-3</sup>
ReeDensityResponse	vertex	$\tilde{Re(n)}$ , <a href="#">AC Response on page 1192</a>	cm <sup>-3</sup> V <sup>-1</sup>
ReeeDiffusionLNVXVSD	vertex	<a href="#">Table 142 on page 816</a>	V <sup>2</sup> scm <sup>-3</sup>
ReeeFlickerGRLNVXVSD	vertex	<a href="#">Table 142 on page 816</a>	V <sup>2</sup> scm <sup>-3</sup>

## Appendix F: Data Names and Plot Names

### Scalar Data

*Table 195 Scalar data (Continued)*

Data name	Location	Description	Unit
ReeeLNVXVSD	vertex	<a href="#">Table 142 on page 816</a>	V <sup>2</sup> scm <sup>-3</sup>
ReeeMonopolarGRLNVXVSD	vertex	<a href="#">Table 142 on page 816</a>	V <sup>2</sup> scm <sup>-3</sup>
ReElectrostaticPotentialResponse	vertex	Re( $\tilde{\phi}$ ) , <a href="#">AC Response on page 1192</a>	1
ReeTemperatureResponse	vertex	Re( $\tilde{T}_n$ ) , <a href="#">AC Response on page 1192</a>	KV <sup>-1</sup>
RefractiveIndex	vertex, element	$n$ , <a href="#">Complex Refractive Index Model on page 694</a>	1
RehDensityResponse	vertex	Re( $\tilde{p}$ ) , <a href="#">AC Response on page 1192</a>	cm <sup>-3</sup> V <sup>-1</sup>
RehhDiffusionLNVXVSD	vertex	<a href="#">Table 142 on page 816</a>	V <sup>2</sup> scm <sup>-3</sup>
RehhFlickerGRLNVXVSD	vertex	<a href="#">Table 142 on page 816</a>	V <sup>2</sup> scm <sup>-3</sup>
RehhLNVXVSD	vertex	<a href="#">Table 142 on page 816</a>	V <sup>2</sup> scm <sup>-3</sup>
RehhMonopolarGRLNVXVSD	vertex	<a href="#">Table 142 on page 816</a>	V <sup>2</sup> scm <sup>-3</sup>
RehTemperatureResponse	vertex	Re( $\tilde{T}_p$ ) , <a href="#">AC Response on page 1192</a>	KV <sup>-1</sup>
ReLatticeTemperatureResponse	vertex	Re( $\tilde{T}$ ) , <a href="#">AC Response on page 1192</a>	KV <sup>-1</sup>
ReLNISD	vertex	<a href="#">Table 142 on page 816</a>	A <sup>2</sup> scm <sup>-3</sup>
ReLNVXVSD	vertex	<a href="#">Table 142 on page 816</a>	V <sup>2</sup> scm <sup>-3</sup>
ReTrapLNISD	vertex	<a href="#">Table 142 on page 816</a>	A <sup>2</sup> scm <sup>-3</sup>
ReTrapLNVSD	vertex	<a href="#">Table 142 on page 816</a>	V <sup>2</sup> scm <sup>-3</sup>

## Appendix F: Data Names and Plot Names

### Scalar Data

Table 195    *Scalar data (Continued)*

Data name	Location	Description	Unit
SemiconductorElectricField	vertex	Electric field $F$ on semiconductor	Vcm <sup>-1</sup>
SpaceCharge	vertex	<a href="#">Equation 37</a>	cm <sup>-3</sup>
SpontaneousRecombination	vertex	<a href="#">Spontaneous Recombination Rate on page 1080</a>	cm <sup>-3</sup> s <sup>-1</sup>
srhRecombination	vertex	$R_{\text{net}}^{\text{SRH}}$ , <a href="#">Shockley–Read–Hall Recombination on page 473</a>	cm <sup>-3</sup> s <sup>-1</sup>
StimulatedRecombination	vertex	<a href="#">Stimulated Recombination Rate on page 1079</a>	cm <sup>-3</sup> s <sup>-1</sup>
StrainedDOSeMassFactor	vertex	$\gamma_n^{2/3}$ , <a href="#">Strained Electron Effective Mass and DOS on page 945</a>	1
StrainedDOShMassFactor	vertex	$\gamma_p^{2/3}$ , <a href="#">Strained Hole Effective Mass and DOS on page 948</a>	1
Stress	vertex	Stress tensor, <a href="#">Chapter 31 on page 935</a>	Pa
StressXX	vertex	Components of stress tensor, <a href="#">Chapter 31 on page 935</a>	Pa
StressXY			
StressXZ			
StressYY			
StressYZ			
StressZZ			

## Appendix F: Data Names and Plot Names

### Scalar Data

*Table 195 Scalar data (Continued)*

Data name	Location	Description	Unit
SurfaceRecombination	vertex	Surface SRH Recombination Model on page 486	$\text{cm}^{-2}\text{s}^{-1}$
ThermalConductivity	vertex	$\kappa$ , Equation 1094	$\text{Wcm}^{-1}\text{K}^{-1}$
ThermalConductivityAniso	vertex	$\kappa_{\text{aniso}}$ , Anisotropic Thermal Conductivity on page 902	$\text{Wcm}^{-1}\text{K}^{-1}$
ThermalizationYield_Bandgap (Plot name: ThermalizationYield)	vertex	Optical Absorption Heat on page 660	1
ThermalizationYield_Vacuum (Plot name: ThermalizationYield)	vertex	Optical Absorption Heat on page 660	1
ThomsonHeat	vertex	Table 28 on page 251	$\text{Wcm}^{-3}$
TotalConcentration	vertex	Specifying Doping Species on page 60	$\text{cm}^{-3}$
TotalCurrentDensity	vertex	$ J_n + J_p + J_D $	$\text{Acm}^{-2}$
TotalHeat	vertex	Sum of all heat generation terms, Chapter 9 on page 245, Temperature in Metals on page 300	$\text{Wcm}^{-3}$
TotalInterfaceTrapConcentration	vertex	Chapter 17 on page 543	$\text{cm}^{-2}$
TotalRecombination	vertex	Sum of all generation–recombination terms, Chapter 16 on page 473	$\text{cm}^{-3}\text{s}^{-1}$
TotalTrapConcentration	vertex	Chapter 17 on page 543	$\text{cm}^{-3}$

## Appendix F: Data Names and Plot Names

### Scalar Data

*Table 195 Scalar data (Continued)*

Data name	Location	Description	Unit
TrapConcentration_* (Plot name: TrapConcPerEntry, Current plot name: TrapConcentrationPerEntry)	vertex	Trap energy level concentrations and integrated trap concentration for each trap entry, see <a href="#">Chapter 17 on page 543</a>	cm <sup>-3</sup>
TrapOccupation_* (Plot name: TrapOccupationPerEntry, Current plot name: TrapOccupationPerEntry)	vertex	Trap energy level occupation for each trap entry, <a href="#">Chapter 17 on page 543</a> ; automatically plotted in TDR file as long as -PlotLoadable is not present in Math section	1
tSRHRecombination	vertex	<a href="#">Dynamic Nonlocal Path Trap-Assisted Tunneling on page 481</a>	cm <sup>-3</sup> s <sup>-1</sup>
ValenceBandEnergy	vertex, element-vertex	$E_V$ , <a href="#">Equation 42</a>	eV
VBBARRIERLowering	vertex	Valence band barrier lowering	eV
VBEnergywithLowering	vertex	Valence band energy modified by barrier lowering	eV
VertexIndex	vertex	Vertex numbers	1
xMoleFraction	vertex	<a href="#">Abrupt and Graded Heterojunctions on page 59</a>	1
yMoleFraction	vertex		1

## Appendix F: Data Names and Plot Names

### Vector Data

## Vector Data

Table 196 Vector data

Data name	Location	Description	Unit
ConductionCurrentDensity	vertex	$\vec{J}_n + \vec{J}_p$ , <a href="#">Equation 57</a> or $\vec{J}_M$ in metals, <a href="#">Equation 142</a>	$\text{Acm}^{-2}$
ContactSurfaceNormal	vertex	Contact surface normal	1
DisplacementCurrentDensity	vertex	$\vec{J}_D$	$\text{Acm}^{-2}$
eCurrentDensity	vertex	$\vec{J}_n$ , <a href="#">Equation 57</a>	$\text{Acm}^{-2}$
eDriftVelocity	vertex	Electron drift velocity	$\text{cm s}^{-1}$
eGradQuasiFermi	vertex	$-\nabla\Phi_n$ , <a href="#">Equation 39</a>	$\text{Vcm}^{-1}$
eHeatFlux	vertex	$\vec{S}_n$ , <a href="#">Equation 79</a>	$\text{Wcm}^{-2}$
ElectricField	vertex, element	$\vec{F}$	$\text{Vcm}^{-1}$
EquilibriumElectricField	vertex	$\vec{F}_{eq}$ , <a href="#">Equation 114</a>	$\text{Vcm}^{-1}$
eSHECurrentDensity	vertex	$\vec{J}_{n, \text{SHE}}$ , <a href="#">Spherical Harmonics Expansion Method on page 853</a>	$\text{Acm}^{-2}$
eSHEVelocity	vertex	$\vec{v}_{n, \text{SHE}}$ , <a href="#">Spherical Harmonics Expansion Method on page 853</a>	$\text{cm s}^{-1}$
eVelocity	vertex	$v_n = \vec{J}_n/qn$	$\text{cm s}^{-1}$
GradPoECImACGreenFunction	vertex	<a href="#">Table 142 on page 816</a>	$\text{VsC}^{-1}\text{cm}^{-1}$
GradPoECReACGreenFunction	vertex	<a href="#">Table 142 on page 816</a>	$\text{VsC}^{-1}\text{cm}^{-1}$
GradPoETImACGreenFunction	vertex	<a href="#">Table 142 on page 816</a>	$\text{A}^{-1}\text{cm}^{-1}$

## Appendix F: Data Names and Plot Names

### Vector Data

Table 196 Vector data (Continued)

Data name	Location	Description	Unit
GradPoETReACGreenFunction	vertex	<a href="#">Table 142 on page 816</a>	A <sup>-1</sup> cm <sup>-1</sup>
GradPoHCImACGreenFunction	vertex	<a href="#">Table 142 on page 816</a>	VsC <sup>-1</sup> cm <sup>-1</sup>
GradPoHCRReACGreenFunction	vertex	<a href="#">Table 142 on page 816</a>	VsC <sup>-1</sup> cm <sup>-1</sup>
GradPoHTImACGreenFunction	vertex	<a href="#">Table 142 on page 816</a>	A <sup>-1</sup> cm <sup>-1</sup>
GradPoHTReACGreenFunction	vertex	<a href="#">Table 142 on page 816</a>	A <sup>-1</sup> cm <sup>-1</sup>
hCurrentDensity	vertex	$\vec{J}_p$ , <a href="#">Equation 57</a>	Acm <sup>-2</sup>
hDriftVelocity	vertex	Hole drift velocity	cm s <sup>-1</sup>
hGradQuasiFermi	vertex	$-\nabla\Phi_p$ , <a href="#">Equation 365</a>	Vcm <sup>-1</sup>
hHeatFlux	vertex	$\vec{S}_p$ , <a href="#">Equation 80</a>	Wcm <sup>-2</sup>
HighFieldEntrancePosition	vertex, element	High-Field Entrance Position and Time on page 520	μm
hSHECurrentDensity	vertex	$\vec{J}_{p, \text{SHE}}$ , <a href="#">Spherical Harmonics Expansion Method on page 853</a>	Acm <sup>-2</sup>
hSHEVelocity	vertex	$\vec{v}_{p, \text{SHE}}$ , <a href="#">Spherical Harmonics Expansion Method on page 853</a>	cm s <sup>-1</sup>
hVelocity	vertex	$v_p = \vec{J}_p / qP$	cm s <sup>-1</sup>
ImConductionCurrentResponse	vertex	$\frac{\dot{z}}{\dot{z}} \frac{\dot{z}}{\dot{z}} \text{Im } J_n + J_p$ AC Current Density Responses on page 1194	Acm <sup>-2</sup> V <sup>-1</sup>
ImDisplacementCurrentResponse	vertex	$\frac{\dot{z}}{\dot{z}} \text{Im } J_D$ AC Current Density Responses on page 1194	Acm <sup>-2</sup> V <sup>-1</sup>

## Appendix F: Data Names and Plot Names

### Vector Data

Table 196 Vector data (Continued)

Data name	Location	Description	Unit
ImeCurrentResponse	vertex	$\vec{z}$ Im $J_n$ AC Current Density <a href="#">Responses on page 1194</a>	$\text{Acm}^{-2}\text{V}^{-1}$
ImeEnFluxResponse	vertex	$\vec{z}$ Im $S_n$ AC Current Density <a href="#">Responses on page 1194</a>	$\text{Wcm}^{-2}\text{V}^{-1}$
ImhCurrentResponse	vertex	$\vec{z}$ Im $J_p$ AC Current Density <a href="#">Responses on page 1194</a>	$\text{Acm}^{-2}\text{V}^{-1}$
ImhEnFluxResponse	vertex	$\vec{z}$ Im $S_p$ AC Current Density <a href="#">Responses on page 1194</a>	$\text{Wcm}^{-2}\text{V}^{-1}$
ImlEnFluxResponse	vertex	$\vec{z}$ Im $S_L$ AC Current Density <a href="#">Responses on page 1194</a>	$\text{Wcm}^{-2}\text{V}^{-1}$
ImTotalCurrentResponse	vertex	$\vec{z}$ $\vec{z}$ $\vec{z}$ Im $J_n + J_p + J_D$ AC Current Density <a href="#">Responses on page 1194</a>	$\text{Acm}^{-2}\text{V}^{-1}$
InsulatorElectricField	vertex	Electric field $\vec{F}$ on insulator	$\text{Vcm}^{-1}$
lHeatFlux	vertex	$\vec{z}$ $S_L$ , <a href="#">Equation 81</a>	$\text{Wcm}^{-2}$
Mod_eGradQuasiFermi_ElectricField	vertex	$\tilde{\nabla\Phi}_n$ , <a href="#">Interpolation of Driving Forces to Zero Field on page 450</a>	$\text{Vcm}^{-1}$
Mod_hGradQuasiFermi_ElectricField	vertex	$\tilde{\nabla\Phi}_p$ , <a href="#">Interpolation of Driving Forces to Zero Field on page 450</a>	$\text{Vcm}^{-1}$

## Appendix F: Data Names and Plot Names

### Vector Data

*Table 196 Vector data (Continued)*

Data name	Location	Description	Unit
NonLocalBackDirection (Plot name: NonLocal)	vertex	Visualizing Nonlocal Meshes on page 203	μm
NonLocalDirection (Plot name: NonLocal)	vertex		μm
OpticalField	vertex	Plot optical intensity as well as real and imaginary parts of optical field	W/m <sup>-3</sup> V/m
PE_Polarization	vertex	$P_{PE}$ , Piezoelectric Datasets on page 1006	Ccm <sup>-2</sup>
Polarization	vertex, element	$\vec{P}$ , Chapter 29 on page 905	Ccm <sup>-2</sup>
ReConductionCurrentResponse	vertex	$\frac{\dot{z}}{\dot{z}} \text{ Re } J_n + J_p$ AC Current Density Responses on page 1194	Acm <sup>-2</sup> V <sup>-1</sup>
ReDisplacementCurrentResponse	vertex	$\frac{\dot{z}}{\dot{z}} \text{ Re } J_D$ AC Current Density Responses on page 1194	Acm <sup>-2</sup> V <sup>-1</sup>
ReeCurrentResponse	vertex	$\frac{\dot{z}}{\dot{z}} \text{ Re } J_n$ AC Current Density Responses on page 1194	Acm <sup>-2</sup> V <sup>-1</sup>
ReeEnFluxResponse	vertex	$\frac{\dot{z}}{\dot{z}} \text{ Re } S_n$ AC Current Density Responses on page 1194	Wcm <sup>-2</sup> V <sup>-1</sup>
RehCurrentResponse	vertex	$\frac{\dot{z}}{\dot{z}} \text{ Re } J_p$ AC Current Density Responses on page 1194	Acm <sup>-2</sup> V <sup>-1</sup>
RehEnFluxResponse	vertex	$\frac{\dot{z}}{\dot{z}} \text{ Re } S_n$ AC Current Density Responses on page 1194	Wcm <sup>-2</sup> V <sup>-1</sup>

## Appendix F: Data Names and Plot Names

### Special Vector Data

*Table 196 Vector data (Continued)*

Data name	Location	Description	Unit
RelEnFluxResponse	vertex	$\vec{z}$ Re $S_L$ <a href="#">AC Current Density Responses on page 1194</a>	$\text{Wcm}^{-2}\text{V}^{-1}$
ReTotalCurrentResponse	vertex	$\vec{z}$ $\vec{z}$ $\vec{z}$ Re $J_n + J_p + J_D$ <a href="#">AC Current Density Responses on page 1194</a>	$\text{Acm}^{-2}\text{V}^{-1}$
SemiconductorElectricField	vertex	Electric field $\vec{F}$ on semiconductor	$\text{Vcm}^{-1}$
TotalCurrentDensity	vertex	$\vec{J}_n + \vec{J}_p + \vec{J}_D$	$\text{Acm}^{-2}$

## Special Vector Data

*Table 197 Special vector data*

Data name	Location	Description	Unit
eSHEDistribution	vertex	Electron energy distribution, <a href="#">SHE Distribution Hot-Carrier Injection on page 852</a>	1
hSHEDistribution	vertex	Hole energy distribution, <a href="#">SHE Distribution Hot-Carrier Injection on page 852</a>	1

## Appendix F: Data Names and Plot Names

### Tensor Data

---

## Tensor Data

Table 198 *Tensor data*

Data name	Location	Description	Unit
ConversePiezoelectricField	vertex	<a href="#">Chapter 31 on page 935</a>	1
ElasticStrain	vertex	Strain tensor from TDR file (Sentaurus Process or Sentaurus Interconnect), <a href="#">Strain Tensor on page 939</a>	1
Strain	vertex	Strain tensor $\bar{\epsilon}$ , <a href="#">Strain and Stress in Semiconductors on page 935</a>	1
Stress	vertex	Stress tensor $\bar{\sigma}$ , <a href="#">Stress and Strain in Semiconductors on page 935</a>	Pa

---

## Particle Data

Table 199 *Particle data*

Plot name	Location	Description	Unit
MIMParticle	MIM defect locations	Particles representing MIM defects, <a href="#">Particle Plots on page 1140</a>	1

# G

## Command File Overview

---

*This appendix presents an overview of the command file of Sentaurus Device.*

[Top Level of Command File on page 1560](#) presents the topmost level of the command file. Use this table to obtain a top-down overview of the command file. The remaining sections are ordered alphabetically. Headings of tables, therein, mostly start with a keyword. Use these tables to find details about keywords.

---

### Organization of Command File Overview

The tables in this appendix have two or three columns, and their rows are ordered alphabetically with respect to the first (and, as far as applicable, the middle) column. Placeholders (cross references, user-supplied values in angle brackets (<>)) precede explicit keywords, irrespective of alphabetic order.

The left column contains keywords that can appear in the command file. In the topmost rows of some tables, the left column references another table. This means that all keywords in the referenced table can appear in the referring table as well. Some keywords are followed by an opening parenthesis or an opening brace to indicate that the keyword expects a selection of options listed in the middle column of the current and the following rows.

The middle column contains options or values to the keyword in the left column, one per row. For a selection of options, the last row indicates the closing parenthesis or the closing brace. For some tables, the middle column would be empty and is omitted.

The right column contains the description of keywords, options, and values of the row. As far as applicable, the right column also provides the following additional information:

- The default value, which can be:
  - An explicit value. Those values are written in Courier font, as you would type them.
  - + or – to indicate the default (true or false, respectively) for keywords that support the – prefix.
  - on or off for keywords that set and reset flags, but do not support the – prefix.

## Appendix G: Command File Overview

### Organization of Command File Overview

- An asterisk (\*) to indicate the default among mutually exclusive alternatives.
- An exclamation mark (!) if no default exists and you must specify a value.
- The unit assumed for the given quantity. In unit specifications,  $d$  denotes the dimension of the mesh. For dimensionless quantities, no unit is specified.
- The location for which keywords should be specified, given as characters in parentheses:
  - (g) stands for global parameters that must not appear in region-specific, interface-specific, or contact-specific sections.
  - (r) stands for region-specific (or material-specific) parameters that usually do not make sense when specified for interfaces or contacts.
  - (c) and (i) stand for contact-specific or interface-specific parameters.

Furthermore, the tables use the following conventions:

- An ellipsis (...) denotes that the preceding item can be repeated an arbitrary number of times.
- An asterisk (\*) followed by an integer, both typeset in Times font, denotes that the preceding item must appear the given number of times.
- Optional components of specifications are enclosed in brackets, and the brackets are *not* part of the syntax.
- Angle brackets (<>) indicate user-supplied values. [Table 200](#) summarizes the common types of specification. More specific notation is explained in the last column of the table where it is used.

Some tables use additional conventions as necessary. They are explained in the text that precedes the respective table.

*Table 200 Notation for user-supplied values*

Specification	Description
<(x,y)> <[x,y]> <(x,y]> <[x,y)>	Range of floating-point numbers from x to y. Parentheses are used when the limits are excluded from the range; brackets are used when they are included. To denote ranges unbound on one side, the corresponding limit is omitted.
<carrier>	A carrier type, Electron or Hole.
<float>	A floating-point number (integers are special cases thereof).

## Appendix G: Command File Overview

### Top Level of Command File

Table 200 Notation for user-supplied values (Continued)

Specification	Description
<ident>	An identifier. This is like a string, but is not enclosed in double quotation marks.
<int>	An integer.
<n..m>	A range of integers, from $n$ to $m$ , inclusively. Either $n$ or $m$ can be omitted, to denote ranges that are unbound on one side.
<string>	A sequence of characters (including digits and special characters) included in double quotation marks. Unlike most Sentaurus Device input, strings are case sensitive.
<vector>	A sequence of up to three floating-point numbers, enclosed in parentheses, and separated by spaces. For example: (1.0 3 0)
<System_Coord>	CrystalSystem or SimulationSystem.

---

## Top Level of Command File

Table 201 lists the top level in the command file of Sentaurus Device.

Table 201 Top level of command file

Table 205		Specification for single devices.
Device	<string>{Table 205}	Device name and device specifications (mixed mode). <a href="#">Device Section on page 110</a>
OpticalDevice	<string>{Table 205}	Optical device name and device specification (mixed mode and <a href="#">Composite Method on page 674</a> ). <a href="#">Device Section on page 110</a>
Solve{	Table 337}	The problem to be solved.
System{	Table 355}	The circuit (mixed mode). <a href="#">System Section on page 103</a>

## Appendix G: Command File Overview

### CurrentPlot

---

## CurrentPlot

Table 202 CurrentPlot{} *Tracking Additional Data in the Current File*

<vector>		μm Print data at coordinate <vertex>.
Average(	<a href="#">Table 203</a> )	Print average over given domain.
Device	=<string>	Name of the device for which the parameter will be plotted.
Integrate(	<a href="#">Table 203</a> )	Print integral over given domain.
Material	=<string>	Name of material for which the parameter will be plotted if it has been specified for a particular material.
MaterialInterface	=<string>	Name of material interface for which the parameter will be plotted if it has been specified for a particular material interface.
Maximum(	<a href="#">Table 203</a> )	Print maximum in given domain.
Minimum(	<a href="#">Table 203</a> )	Print minimum in given domain.
Model	=<string>	Select a model for printing of parameters.
ModelProperty	=<string>	Select model parameter for plotting when using unified interface for optical generation computation. <a href="#">Parameter Ramping on page 689</a>
OpticalDevice	=<string>	Name of the optical device for which the parameter will be plotted.
Parameter	=<string>	Print parameter <string>.
Region	=<string>	Name of the region for which the parameter will be plotted if it has been specified for a particular region.
RegionInterface	=<string>	Name of the region interface for which the parameter will be plotted if it has been specified for a particular region interface.
Tcl(	<a href="#">Table 204</a> )	Evaluate a Tcl formula. <a href="#">Tcl Formulas on page 170</a>

## Appendix G: Command File Overview

### CurrentPlot

**Table 203** *Average(), Integrate(), Maximum(), and Minimum() Tracking Additional Data in the Current File*

Table 362		Sample over location.
Coordinates		Print coordinates where maximum, minimum, or average occurs.
DopingWell	<vector>	Sample over well, defined by point in the well.
Everywhere		Sample over entire device.
Insulator		Sample over all insulator regions of the device.
Name	=<string>	Name for sampled data to be used in output.
Semiconductor		Sample over all semiconductor regions of the device.
Window[	<vector> <vector>]	Sample over the window, defined by two specified corners.

**Table 204** *Tcl() Tcl Formulas*

<ident>	=<string>   <float>	User-specific model parameters passed to an external Tcl current plot script. <a href="#">Current Plot File on page 1491</a>
Dataset	=<string>	Dataset name that appears in the header section of the current plot file. <a href="#">Dataset Option on page 171</a>
Finish	=<string>	Tcl code executed last for each plot time point. <a href="#">Finish Option on page 173</a>
Formula	=<string>	Tcl code executed on each mesh vertex. <a href="#">Formula Option on page 172</a>
Function	=<string>	Function name that appears in the header section of the current plot file. <a href="#">Function Option on page 172</a>
Init	=<string>	Tcl code executed first for each plot time point. <a href="#">Init Option on page 172</a>
Operation	=<string>	Sets the operation mode, that is, how the results of each vertex are combined. <a href="#">Operation Option on page 173</a>

## Appendix G: Command File Overview

### Device

Table 204 *Tcl()* *Tcl Formulas* (Continued)

Tcl	= "source <ident>"	Tcl code defined in external file for more complex tasks, where <ident> refers to the Tcl script file. <a href="#">Current Plot File on page 1491</a>
Unit	=<string>	Unit of the current plot quantity. <a href="#">Unit Option on page 172</a>

---

## Device

Table 205 *Device{}}, *OpticalDevice{}}, or single-device specification* [Chapter 2 on page 57](#)*

CurrentPlot{	<ident>	Use current plot PMI. <a href="#">Current Plot File on page 1439</a>
	PMIModel ( <a href="#">Table 324</a> )	Use current plot PMI with parameters. <a href="#">Current Plot File on page 1439</a>
	<a href="#">Table 195</a> ( <a href="#">Table 202</a> )	Plot scalar data to current file. <a href="#">Tracking Additional Data in the Current File on page 165</a>
	<a href="#">Table 196</a> / Vector( <a href="#">Table 202</a> )}	Plot vectorial data to current file. <a href="#">Tracking Additional Data in the Current File on page 165</a>
Electrode{	<a href="#">Table 206</a> }	Electrode specification. <a href="#">Specifying Electrical Boundary Conditions on page 115</a>
eSHEDistributionPlot{	<vector>... }	Electron-energy distribution plot. <a href="#">Visualizing Spherical Harmonics Expansion Method on page 866</a>
Extraction{	<ident> [=] <ident>... }	Process parameters. <a href="#">Extraction File on page 184</a>
File{	<a href="#">Table 207</a> }	Input and output files.
GainPlot{	<a href="#">Table 208</a> }	Parameters for gain plot (LED).
hSHEDistribution Plot{	<vector>... }	Hole-energy distribution plot. <a href="#">Visualizing Spherical Harmonics Expansion Method on page 866</a>
HydrogenBoundary{	<a href="#">Table 209</a> }	Degradation model. <a href="#">Boundary Conditions on page 613</a>

## Appendix G: Command File Overview

### Device

*Table 205 Device{}, OpticalDevice{}, or single-device specification Chapter 2 on page 57*

IFM{	<a href="#">Table 210</a> }	Variations, noise models. <a href="#">IFM Section on page 812</a>
Math	[ ( <a href="#">Table 359</a> ) ]	Math specification, potentially restricted to a location.
{	<a href="#">Table 211</a> }	
MonteCarlo{	<options>}	See the <i>Sentaurus™ Device Monte Carlo User Guide</i>
NoisePlot{	<a href="#">Table 234</a> }	Noise output data. <a href="#">Noise Output Data on page 815</a>
NonLocalPlot	(<vector>...) { <a href="#">Table 235</a> }	Nonlocal plot. <a href="#">Visualizing Data Defined on Nonlocal Meshes on page 204</a>
Physics	[ ( <a href="#">Table 359</a> ) ]	Physical models, potentially restricted to a location.
{	<a href="#">Table 236</a> }	
Plot{	<a href="#">Table 334</a> }	Plot data. <a href="#">Device Plots on page 177</a>
RayTraceBC{	<a href="#">Table 336</a> }	Raytrace boundary conditions.
TensorPlot(	<a href="#">Table 357</a> )	Tensor plot for beam propagation method. <a href="#">Visualizing Results on Native Tensor Grid on page 764</a>
{	ComplexRefractive Index	
	OpticalField	
	OpticalIntensity}	
Thermode{	<a href="#">Table 358</a> }	Thermode specification. <a href="#">Specifying Thermal Boundary Conditions on page 118</a>
TrappedCarDistrPlot{	<vector>	Trapped carrier charge plot at position <vector>. <a href="#">Visualizing Traps on page 578</a>
	<a href="#">Table 362={&lt;vector&gt;...}}</a>	Trapped carrier charge plot restricted to location.

## Appendix G: Command File Overview

### Electrode

## Electrode

Table 206 *Electrode{} Specifying Electrical Boundary Conditions*

AreaFactor	=<float>	1 Multiplier for electrode current. <a href="#">Reading a Structure on page 57</a>
Barrier	=<float>	V Barrier voltage.
Charge	=<float>	C Charge for floating electrode. Voltage must not be specified. <a href="#">Floating Contacts on page 280</a>
	=(<float1> At <float2>[, , ...])	C, s Transient charge behavior. A list of pairs of charges <float1> at time <float2>, piecewise linearly interpolated.
Current	=<float>	A $\mu$ m <sup>d-3</sup> Current boundary condition. Voltage is used as initial guess only.
	=(<float1> At <float2>[, , ...])	A $\mu$ m <sup>d-3</sup> , s Transient current behavior. A list of pairs of currents <float1> at time <float2>, piecewise linearly interpolated.
CyclicNorm	=<string>	Output of cyclic extrapolation. File stores information about cyclic norm versus number of cyclic cycles and time.
DistResist	=<float>	$\Omega$ cm <sup>2</sup> Distributed resistance. <a href="#">Resistive Contacts on page 270</a>
	=SchottkyResist	Use Schottky contact resistance model. <a href="#">Resistive Contacts on page 270</a>
EqOhmic		off Contact is Ohmic with a modified boundary condition. <a href="#">Modified Ohmic Contacts on page 259</a>
eRecVelocity	=<[ 0 , )>	2.573e6 cms <sup>-1</sup> Electron recombination velocity. <a href="#">Schottky Contacts on page 261</a>

## Appendix G: Command File Overview

### Electrode

**Table 206    Electrode{} Specifying Electrical Boundary Conditions (Continued)**

Extraction{	bulk	Specify electrode as bulk for extraction purposes. <a href="#">Extraction File on page 184</a>
	drain	Specify electrode as drain for extraction purposes. <a href="#">Extraction File on page 184</a>
	gate	Specify electrode as gate for extraction purposes. <a href="#">Extraction File on page 184</a>
	source}	Specify electrode as source for extraction purposes. <a href="#">Extraction File on page 184</a>
FGcap=(	value=<float>	F $\mu$ m <sup>d-3</sup> Additional capacitance for floating electrode. <a href="#">Floating Metal Contacts on page 280</a>
	name=<string>)	Coupling to electrode <string>. <a href="#">Floating Metal Contacts on page 280</a>
hRecVelocity	=<[ 0 , )>	1.93e6 cms <sup>-1</sup> Hole recombination velocity. <a href="#">Schottky Contacts on page 261</a>
Material	=<string>	Electrode material. <a href="#">Contacts on Insulators on page 260</a>
	=<string> (N=<( 0 , )>)	cm <sup>-3</sup> Electrode material with n-doping. <a href="#">Contacts on Insulators on page 260</a>
	=<string> (P=<( 0 , )>)	cm <sup>-3</sup> Electrode material with p-doping. <a href="#">Contacts on Insulators on page 260</a>
Name	=<string>	Name of electrode.
	=Regexp(<string>)	Regular expression for matching structure contacts.
Poisson	=Dirichlet	* Use Dirichlet-like boundary condition for Poisson equation.
	=Neumann	Use homogeneous Neumann boundary condition for Poisson equation at Ohmic contacts.
Resist	=<float>	$\Omega\mu$ m <sup>3-d</sup> Contact resistance. <a href="#">Resistive Contacts on page 270</a>
Schottky		off Contact is a Schottky contact. <a href="#">Schottky Contacts on page 261</a>

## Appendix G: Command File Overview

### File

Table 206 *Electrode{ Specifying Electrical Boundary Conditions (Continued)*

SRDoping		Include the image force lowering of the potential energy barrier at Schottky contact. <a href="#">Resistive Contacts on page 270</a>
Voltage	=<float>	V Contact voltage.
	=(<float1> At <float2>[, , ] . . .)	V, s Transient voltage behavior. A list of pairs of voltages <float1> at time <float2>, piecewise linearly interpolated.
Workfunction	=<float>	eV Electrode workfunction. <a href="#">Contacts on Insulators on page 260</a>

---

## File

In the description of [Table 207](#), it is indicated whether a file is input or output, as well as the extensions (in Courier font) that Sentaurus Device appends to the user-supplied name to obtain the full name.

The tags enclosed in at-signs (@) denote the default Sentaurus Workbench variables for the particular file name. (These defaults can be changed in `tooldb.tcl`, see [Sentaurus™ Workbench User Guide](#), Global Configuration Files. (g) denotes keywords that must be used in global `File` sections only, while (d) denotes keywords that must be used in device-specific `File` sections only.

Table 207 *File{*

ACEextract	=<string>	(g) Small-signal and noise analysis (output, <code>_ac_des.plt</code> , <code>@acplot@</code> ). <a href="#">Small-Signal AC Analysis on page 149</a>
Bandstructure	=<string>	(d) Base name for plotting local band structure data in an LED simulation (output, <code>_kpbandstruc_vertexX_des.plt</code> , <code>_kpeigenfunc_vertexX_des.plt</code> ).
CarrierPath	=<string>	Carrier path and spatially interpolated plot variables (output <code>.tdr</code> ). <a href="#">Save Carrier Paths as 1D Datasets in Separate TDR File on page 522</a>

## Appendix G: Command File Overview

### File

*Table 207 File{} (Continued)*

CMIPath	=<string>	(g) Search path for compact circuit files (extension .ccf) and the corresponding shared object files (extension .so.arch). The files are parsed and added to the System section of the command file. If the environment variable STROOT_ARCH_OS_LIB is defined, the directory \$STROOT_ARCH_OS_LIB/sdevice is automatically added to CMIPath. <a href="#">System Section on page 103</a>
Current	=<string>	Device currents, voltages, charges, temperatures, and times (output, _des.plt, @plot@). <a href="#">Current File on page 162</a>
DephasingRates	=<string>	Directory for saving dephasing rates using the second Born approximation.
DevFields	=<string>	(d) Space distribution for trapped carrier charge density. Must match grid file (input .tdr). <a href="#">Energetic and Spatial Distribution of Traps on page 544</a>
DevicePath	=<string>	(g) Load all files with the extension .device in the directory path <string>. The directory path has the format dir1:dir2:dir3. The devices found can be used in the System section. They are not overwritten by a definition with the same name in the command file. <a href="#">System Section on page 103</a>
EmissionTable	=<string>	Tabulated emission data (output).
eSHEDistribution	=<string>	Electron distribution versus kinetic energy (output, _des.plt). <a href="#">Visualizing Spherical Harmonics Expansion Method on page 866</a>
Extraction	=<string>	(d) Extraction file (output, extraction_des.xtr). <a href="#">Extraction File on page 184</a>
Gain	=<string>	(d) Modal stimulated and spontaneous emission spectra (output, _gain_des.plt).
Grid	=<string>	(d) Device geometry and mesh (input, .tdr, or @tdr@).
hSHEDistribution	=<string>	Hole distribution versus kinetic energy (output, _des.plt). <a href="#">Visualizing Spherical Harmonics Expansion Method on page 866</a>
IlluminationSpectrum	=<string>	Illumination spectrum. <a href="#">Illumination Spectrum on page 651</a>

## Appendix G: Command File Overview

### File

*Table 207 File{} (Continued)*

IonIntegral	=<string>	Breakdown path and spatially interpolated simulation results along the breakdown path (output, _des_ABA.tdr). <a href="#">Visualizing Breakdown Paths on page 515</a>
LifeTime	=<string>	(d) Lifetime profiles (input, .tdr). <a href="#">Lifetime Profiles From Files on page 475</a>
Load	=<string>	Old simulation results (input, .sav). <a href="#">Save and Load Statements on page 213</a>
MesherInput	=<string>	Base name of boundary and command file to be used for calling Sentaurus Mesh. <a href="#">Specifying the Optical Solver on page 669</a>
MIMBand	=<string>	Metal–insulator–metal (MIM) band diagram (output, _mimband_des.tdr). <a href="#">Band Diagram With Traps on page 1140</a>
MIMCurrent	=<string>	MIM steady-state currents (output, _mimcur_des.plt). <a href="#">Special MIM Current File (Currents and Capacitance) on page 1139</a>
MIMSensitivity	=<string>	MIM sensitivity analysis (output, _<volt>_mimsa_des.plt). <a href="#">Sensitivity Analysis on page 1125</a>
MobilityDoping	=<string>	(d) File from which donor and acceptor concentrations for mobility calculations are read. <a href="#">Mobility Doping File on page 465</a>
NewtonPlot	=<string>	Convergence monitoring (output). <a href="#">NewtonPlot on page 209</a>
NonLocalPlot	=<string>	Data defined on nonlocal line meshes (output). <a href="#">Visualizing Data Defined on Nonlocal Meshes on page 204</a>
OpticalGenerationFile	=<string>	Load optical generation from a file. <a href="#">Optical AC Analysis on page 770</a>
OpticalGenerationInput	=<string>	File from which optical generation rate is loaded. <a href="#">Loading and Saving Optical Generation From and to Files on page 656</a>
OpticalGenerationOutput	=<string>	File to which optical generation rate is written. <a href="#">Loading and Saving Optical Generation From and to Files on page 656</a>

## Appendix G: Command File Overview

### File

*Table 207 File{} (Continued)*

OpticalGenerationOutput (collected)	=<string>	File to which optical generation rate is written. Consecutive plots are written into one file instead of several enumerated files. <a href="#">Loading and Saving Optical Generation From and to Files on page 656</a>
OpticalSolverInput	=<string>	Command file of optical solver. <a href="#">Specifying the Optical Solver on page 669</a>
OpticsOutput	=<string>	(g) Runtime log for optical solvers (output, _des.log, @optlog@).
Output	=<string>	"output" (g) Runtime log (output, _des.log, @log@).
ParameterPath	=<string>	(d) List of subdirectories in the \$STROOT/tcad/\$STRELEASE/lib/sdevice/MaterialDB directory that are added to the search path for parameter files. <a href="#">Physical Model Parameters on page 70</a>
Parameters	=<string>	(d) Device parameters (input, .par, @parameter@). <a href="#">Physical Model Parameters on page 70</a>
Piezo	=<string>	(d) Stress data for piezoelectric model (input). <a href="#">Chapter 31 on page 935</a>
Plot	=<string>	Spatially distributed simulation results (output, _des.tdr, @tdrdat@). <a href="#">Device Plots on page 177</a>
Plot(collected)	=<string>	Spatially distributed simulation results (output, _des.tdr, @tdrdat@). Consecutive plots are written into one file instead of several enumerated files. <a href="#">Device Plots on page 177</a>
PMIPath	=<string>	(g) Search path to load shared object files (extension .so.arch) for PMI models. <a href="#">Command File of Sentaurus Device on page 1230</a>
PMIUserFields	=<string>	(d) File containing any of the fields PMIUserField0 to PMIUserField299.
SpectralPlot	=<string>	Spatially distributed simulation results (output, _des.tdr, @spectralplot@) for each entry of the illumination spectrum as well as spectral curves (_des.plt) for the results of the optical solver. Consecutive plots are written into one file instead of several enumerated files, that is, SpectralPlot(collected) is the default. <a href="#">Illumination Spectrum on page 651</a>

## Appendix G: Command File Overview

### GainPlot

Table 207 *File{}* (Continued)

SpectralPlot(-collected)	=<string>	Spatially distributed simulation results (output, <code>_des.tdr</code> , @spectralplot@) for each entry of the illumination spectrum as well as spectral curves ( <code>_des.plt</code> ) for the results of the optical solver. Consecutive plots are written into several enumerated files. <a href="#">Illumination Spectrum on page 651</a>
TensorPlot	=<string>	Base name for saving tensor plot files when using beam propagation method. <a href="#">Visualizing Results on Native Tensor Grid on page 764</a>
Save	=<string>	Simulation results for retrieval with <code>Load</code> (output, <code>_des.sav</code> ). <a href="#">Save and Load Statements on page 213</a>
SaveOptField	=<string>	Base name for saving optical field data in an LED simulation.
SPICEPath	=<string>	Search path for SPICE circuit files (extension <code>.scf</code> ). The files are parsed and added to the <code>System</code> section of the command file. If the environment variable <code>STROOT_LIB</code> is defined, the directory <code>\$STROOT_LIB/sdevice/spice</code> is automatically added to <code>SPICEPath</code> . <a href="#">File Section on page 110</a>
TrappedCarPlotFile	=<string>	Trapped carrier charge density, trap occupancy probability, and trap density versus energy (output, <code>_des.plt</code> ). <a href="#">Visualizing Traps on page 578</a>

---

## GainPlot

Table 208 *GainPlot{}*

Intervals	=<int>	Number of points to plot for each gain curve.
Range	=(<float>*2)	eV Energy range of the gain plot.
	=Auto	Automatically find the energy range.

## Appendix G: Command File Overview

### HydrogenBoundary

---

## HydrogenBoundary

Table 209    *HydrogenBoundary{ } Boundary Conditions*

HydrogenAtom	=<float>	cm <sup>-3</sup> Boundary concentration.
	=reflective	Homogeneous Neumann boundary condition
HydrogenIon	=<float>	cm <sup>-3</sup> Boundary concentration.
	=reflective	Homogeneous Neumann boundary condition.
HydrogenMolecule	=<float>	cm <sup>-3</sup> Boundary concentration.
	=reflective	Homogeneous Neumann boundary condition.
HydrogenSpeciesA	=<float>	cm <sup>-3</sup> Boundary concentration.
	=reflective	Homogeneous Neumann boundary condition.
HydrogenSpeciesB	=<float>	cm <sup>-3</sup> Boundary concentration.
	=reflective	Homogeneous Neumann boundary condition.
HydrogenSpeciesC	=<float>	cm <sup>-3</sup> Boundary concentration.
	=reflective	Homogeneous Neumann boundary condition.
Name	=<string>	Name of the hydrogen contact.

---

## IFM

Table 210    *IFM{ } IFM Section*

DeterministicVariation(	<a href="#">Table 290</a> )	(g) Deterministic variations. <a href="#">Deterministic Doping Variations on page 809</a>
Noise	[<string>] ( <a href="#">Table 319</a> )	(r) Noise sources. <a href="#">Noise Sources on page 787</a>
RandomizedVariation	<string> ( <a href="#">Table 325</a> )	(g) Set sIFM models. <a href="#">Statistical Impedance Field Method on page 797</a>

## Appendix G: Command File Overview

### Math

**Table 211** *Math{}*

<a href="#">Table 232</a>		Transient time-step control.
AcceptNewtonParameter(	<a href="#">Table 212</a>	(g) Relaxed Newton method. <a href="#">Relaxed Newton Method on page 198</a>
ACMethod	=Blocked	* Use block decomposition solver for ACCoupled.
ACSubMethod	= <a href="#">Table 223</a>	Inner linear solver for Blocked method for ACCoupled.
AllowLayerThickness	=Everywhere	(g) Account for LayerThickness in all materials. <a href="#">Extracting Layer Thickness on page 379</a>
	=Insulator	(g) Account for LayerThickness in insulators only.
	=Metal	(g) Account for LayerThickness in metals only.
	=Semiconductor	* (g) Account for LayerThickness in semiconductors only.
AnisoGradQF_formula	=0   1	0 Select formula for driving force. <a href="#">Driving Forces on page 895</a>
AnisoSG		off (g) Use anisotropic Scharfetter–Gummel approximation for anisotropic models. <a href="#">Chapter 28 on page 883</a>
AnisoSG_DerivativeMinDen	=<float>	0.0 cm <sup>-3</sup> (g) Lower limit of carrier densities for AnisoSG approximation. Corresponding derivatives are switched off if the node concentration is less than this value. <a href="#">Chapter 28 on page 883</a>

## Appendix G: Command File Overview

### Math

*Table 211 Math{} (Continued)*

AnisoSG_MinDen	=<float>	10 cm <sup>-3</sup> (g) Stabilization parameter for computation of $x^*$ for AnisoSG approximation. <a href="#">Chapter 28 on page 883</a>
AutoCNPMinStepFactor		2.0 (g) Multiplier of MinStep for the automatic activation of CNormPrint. <a href="#">Automatic Activation of CNormPrint and NewtonPlot on page 210</a>
AutomaticCircuitContact		on Poisson covers the circuit. <a href="#">Additional Equations Available in Mixed Mode on page 197</a>
AutoNPMMinStepFactor		2.0 (g) Multiplier of MinStep for the automatic activation of NewtonPlot. <a href="#">Automatic Activation of CNormPrint and NewtonPlot on page 210</a>
AutoOrientation	=(<int>...)	Miller indices for surface orientations supported by AutoOrientation. <a href="#">Changing Orientations Used With Auto-Orientation on page 87</a>
AutoOrientationSmoothingDistance	=<float>	0.0 μm (r) Auto-orientation smoothing distance. <a href="#">Auto-Orientation Smoothing on page 87</a>
AvalDensGradQF		- Alternative driving force. <a href="#">Using Avalanche Generation on page 495</a>
AvalDerivatives		+ Compute analytic derivatives of avalanche generation. <a href="#">Derivatives on page 196</a>
AvalFlatElementExclusion	=<float>	0.0 degree (g) Exclude elements from contributing to avalanche generation. <a href="#">Using Avalanche Generation on page 495</a>

## Appendix G: Command File Overview

### Math

*Table 211 Math{} (Continued)*

AvalPostProcessing		off Impact ionization-generated carriers are not included self-consistently in the solution. <a href="#">Approximate Breakdown Analysis With Carriers on page 517</a>
AverageAniso		(g) Use average-aniso approximation for anisotropic models. <a href="#">Chapter 28 on page 883</a>
AverageBoxMethod		+ Use element-oriented element intersection box method algorithm. If disabled, use quadrilateral box method algorithm. <a href="#">Box Method Coefficients in the 3D Case on page 1179</a>
BehavSrcNumericDeriv	=<float>	o Compute the numeric Jacobian.
BM_ExtendedPrecision		+ Use “long double” precision to compute box method coefficients and control volumes. <a href="#">Extended Precision on page 1179</a>
-BM_StableCalculation		+ Use improved and more accurate algorithms to compute box method coefficients and control volumes. <a href="#">Improved Accuracy of Box Method Parameters on page 1191</a>
BM_VoronoiRISurfaces3D		- Use improved discretization of region interfaces for distributed resistance simulations. <a href="#">Resistive Interfaces on page 277</a>
BoxCoefficientsFromFile	[ (GrdNumbering) ]	Try to read sections of the geometry file. <a href="#">Saving and Restoring Box Method Coefficients on page 1187</a>
BoxMeasureFromFile	[ (GrdNumbering) ]	Try to read sections of the geometry file. <a href="#">Saving and Restoring Box Method Coefficients on page 1187</a>

## Appendix G: Command File Overview

### Math

*Table 211 Math{} (Continued)*

BoxMethodFromFile		+ Read Voronoï surface from this file if the grid file has a <code>VoronoiFaces</code> section. <a href="#">Box Method Coefficients in the 3D Case on page 1179</a>
BreakAtIonIntegral	(<int> <float>)	1 1 Terminate the quasistationary simulation when the <int> largest ionization integrals are greater than <float>. <a href="#">Approximate Breakdown Analysis on page 511</a>
BreakCriteria{	<a href="#">Table 213</a> }	Break criteria. <a href="#">Break Criteria: Conditionally Stopping the Simulation on page 119</a>
BroadeningIntegration (GaussianQuadrature(Order	=<int>))	Use Gaussian quadrature to integrate the broadening spectrum of an LED simulation. <a href="#">Accelerating Gain Calculations and LED Simulations on page 1047</a>
CarrierDensityInAltermatt		+ Use carrier density in the Altermatt ionization model. <a href="#">Physical Model Parameters on page 343</a>
CDensityMin		0 . 0 Acm <sup>-2</sup> (r) Current limit for parallel electric-field computation. <a href="#">Electric Field Parallel to the Interface on page 448</a>
CheckRhsAfterUpdate		– Check whether the RHS can be reduced further after the update error has converged. <a href="#">Convergence and Error Control on page 193</a>
CheckUndefinedModels		+ (g) Check for undefined physical parameters. <a href="#">Undefined Physical Models on page 82</a>
CNormPrint		(g) Convergence monitoring. <a href="#">CNormPrint on page 209</a>

## Appendix G: Command File Overview

### Math

*Table 211 Math{} (Continued)*

CompatibleFEPolarization		Previous formulation of Ginzburg–Landau–Khalatnikov equation. <a href="#">Multidimensional on page 916</a>
ComputeDopingConcentration		– (g) Recompute net doping based on individual doping species. <a href="#">Specifying Doping Species on page 60</a>
ComputeGradQuasiFermiAt Contacts	=UseElectrostatic Potential	* Use the electrostatic potential to compute the GradQuasiFermi driving force within elements touching a contact. <a href="#">Equation 366 on page 448</a>
	=UseQuasiFermi	Use the quasi-Fermi potential to compute the GradQuasiFermi driving force within elements touching a contact. <a href="#">Equation 365 on page 448</a>
ComputeIonizationIntegrals		off Compute ionization integrals using old discretization scheme. <a href="#">Approximate Breakdown Analysis on page 511</a>
	(Table 215)	
ConstRefPot	=<float>	eV Value for $\phi_{ref}$ . <a href="#">Quasi-Fermi Potential With Boltzmann Statistics on page 231</a>
CoordinateSystem{	Table 1}	(g) Coordinate system of explicit coordinates in command file. <a href="#">Reading a Structure on page 57</a>
cT_Range	=(<float*2)	10 80000 K (g) Lower and upper limit for carrier temperature. <a href="#">Numeric Parameters for Temperature Equations on page 256</a>

## Appendix G: Command File Overview

### Math

*Table 211 Math{} (Continued)*

CurrentPlot(	Digits=<float>	6 Number of digits in the names of quantities in the current plot file. <a href="#">CurrentPlot Options on page 169</a>
	IntegrationUnit= <string>	um Length unit for current plot integration. <a href="#">CurrentPlot Options on page 169</a>
CurrentWeighting		off Compute contact currents using an optimal weighting scheme. <a href="#">Numeric Approaches for Contact Current Computation on page 242</a>
Cylindrical	(<float>)	off Use the 2D mesh to simulate a 3D cylindrical device. The device is assumed to be rotationally symmetric around the vertical axis given by $x = <\text{float}> \mu\text{m}$ . $<\text{float}>$ must be less than or equal to the smallest horizontal device coordinate, and defaults to 0. <a href="#">Reading a Structure on page 57</a>
	(xAxis=<float>)	This is same as (<float>).
	(yAxis=<float>)	The device is assumed to be rotationally symmetric around the horizontal axis given by $y = <\text{float}> \mu\text{m}$ . $<\text{float}>$ must be less than or equal to the smallest vertical device coordinate. For UCS coordinates, the x-axis is the vertical axis, so you must specify the cylindrical axis using yAxis.
DeltaVolumeLimit	=<float>	1e-4 Limit for DeltaVolume. <a href="#">Region Non-Delaunay Elements on page 1189</a>
DensityIntegral	(<int>)	30 (g) Defines a number of Gauss–Laguerre quadrature integration points. <a href="#">Using Multivalley Band Structure on page 331</a>

## Appendix G: Command File Overview

### Math

*Table 211 Math{} (Continued)*

DensLowLimit	=<float>	1e-100 cm <sup>-3</sup> (g) Lower limit for carrier densities. <a href="#">Numeric Parameters for Continuity Equation on page 241</a>
DensLowLimitPower	=<int>	-100 (g) Lower limit power for carrier densities ( $10^{\text{DensLowLimitPower}}$ cm <sup>-3</sup> ). <a href="#">Numeric Parameters for Continuity Equation on page 241</a>
Derivatives		+ Compute analytic derivatives of mobility and avalanche generation. <a href="#">Derivatives on page 196</a>
Digits	=<float>	5 Approximate number of digits to which equations must be solved to be considered as converged. <a href="#">Convergence and Error Control on page 193</a>
DirectCurrent		off Compute contact currents directly, using only contact nodes and their neighbors. <a href="#">Numeric Approaches for Contact Current Computation on page 242</a>
DirectQuantumCorrection		off Use the quantum potential directly as the band edge. <a href="#">Notes on the Use of the Density Gradient Model on page 367</a>
DualGridInterpolation	(Method=Conservative)	Use conservative element-based interpolation for dual-grid simulations. <a href="#">Dual-Grid Setup for Raytracing on page 732</a> , <a href="#">Single-Grid Versus Dual-Grid LED Simulation on page 1039</a>
	(Method=Simple)	Use bilinear vertex-based interpolation for dual-grid simulations. <a href="#">Dual-Grid Setup for Raytracing on page 732</a> , <a href="#">Single-Grid Versus Dual-Grid LED Simulation on page 1039</a>

## Appendix G: Command File Overview

### Math

*Table 211 Math{} (Continued)*

eB2BGenWithinSelectedRegions		+ (g) Restrict nonlocal path band-to-band electron generation to regions where model is active. <a href="#">Dynamic Nonlocal Path Band-to-Band Tunneling Model on page 529</a>
eDrForceRefDens	=< [ 0 , ) >	0 cm <sup>-3</sup> (r) Damping parameter for high-field mobility driving force, alias for RefDens_eGradQuasiFermi_Zero. <a href="#">Interpolation of Driving Forces to Zero Field on page 450</a>
ElementEdgeCurrent		- (g) Use an alternative element-edge approximation of the current density compared to the default edge approximation.
ElementVolumeAvalanche		- (g) Use truncated element-vertex volumes for avalanche generation. <a href="#">Using Avalanche Generation on page 495</a>
eMobilityAveraging	=Element	* (r) Use element averaged electron mobility. <a href="#">Mobility Averaging on page 464</a>
	=ElementEdge	(r) Use element-edge averaged electron mobility. <a href="#">Mobility Averaging on page 464</a>
EnormalInterface(	MaterialInterface= [<string>...]	Material interfaces for normal electric field computation. <a href="#">Normal to the Interface on page 426</a>
	RegionInterface= [<string>...])	Region interfaces for normal electric field computation. <a href="#">Normal to the Interface on page 426</a>

## Appendix G: Command File Overview

### Math

*Table 211 Math{} (Continued)*

EparallelToInterface(		Options for EparallelToInterface driving force. <a href="#">Electric Field Parallel to the Interface on page 448</a>
	Direction=<vector>	! Direction of the current.
	Box=( <vector> <vector> ) )	Restrict Direction vector to vertices contained in box.
eQPBox(	QBoxForBC	Apply the box to select boundary condition on interfaces.
	<a href="#">Table 216</a> )	Quantum box for eQuantumPotential. <a href="#">Density Gradient Model on page 362</a>
EquilibriumSolution(	<a href="#">Table 217</a> )	(g) Parameters for equilibrium solution used in specific models. <a href="#">Equilibrium Solution on page 231</a>
Error	( <a href="#">Table 342</a> )=<float>	Value of $\varepsilon_A$ . <a href="#">Convergence and Error Control on page 193</a>
ErrRef	( <a href="#">Table 342</a> )=<float>	Value of $x_{ref}$ . <a href="#">Convergence and Error Control on page 193</a>
ExcludeTouchingContactParts		(g) off Allow simulation for device geometry with touching contacts.
ExitOnFailure		(g) off Terminate the simulation as soon as a <code>Solve</code> command fails.
ExitOnUnknownParameterRegion		+ (g) Exit when unknown region, region interface, or electrode appears in parameter file.
-ExitOnUnknownParameterRegion		off (g) Print warning message when unknown region, region interface, or electrode appears in parameter file.
ExtendedPrecision	[ ( <a href="#">Table 26</a> ) ]	(g) off Use extended precision floating-point arithmetic. <a href="#">Extended Precision on page 223</a>

## Appendix G: Command File Overview

### Math

*Table 211 Math{} (Continued)*

Extrapolate	( <a href="#">Table 218</a> )	– Use extrapolation to obtain an initial guess for the next solution based on the previous solutions. <a href="#">Extrapolation on page 130</a> , <a href="#">Extrapolation on page 143</a>
FEPolarizationIP		Divergence-free polarization constraint. <a href="#">Using the Ginzburg–Landau–Khalatnikov Equation on page 911</a>
FEPolarizationIP	=<float>	0 Numeric parameter for imposing divergence-free polarization constraint. <a href="#">Chapter 29 on page 905</a>
FEPolarizationSolver2		off Use the improved solver for FEPolarization equation. <a href="#">Solver II on page 917</a>
FermiBandTailStatistics		+ Use Fermi–Dirac statistics for band tail density. <a href="#">Band Tails on page 325</a>
GeometricDistances		+ (g) Use enhanced distance and normal to interface computation for mobility and MLDA. <a href="#">Normal to the Interface on page 426</a>
GrainFieldGenerator	( <a href="#">Table 219</a> )	(r) Generate grain field. <a href="#">Creating the Grain Field on page 1110</a>
HB	{ <a href="#">Table 220</a> }	Harmonic Balance on page 153; not device specific.
HBPlotFilePerNewton		- (g) Plot harmonic balance .plt files per converged solve. <a href="#">Circuit Currents and Voltages on page 158</a>
hDrForceRefDens	=<[ 0 , )>	0 cm <sup>-3</sup> (r) Damping parameter for high-field mobility driving force, alias for RefDens_hGradQuasiFermi_Zero. <a href="#">Interpolation of Driving Forces to Zero Field on page 450</a>

## Appendix G: Command File Overview

### Math

*Table 211 Math{} (Continued)*

hMobilityAveraging	=Element	(r) Use element averaged hole mobility. <a href="#">Mobility Averaging on page 464</a>
	=ElementEdge	* (r) Use element-edge averaged hole mobility. <a href="#">Mobility Averaging on page 464</a>
hQPBox(	QBoxForBC	Apply the box to select boundary condition on interfaces.
	Table 221)	Quantum box for hQuantumPotential. <a href="#">Density Gradient Model on page 362</a>
IgnoreTdrUnits		off (g) Ignore TDR units when loading data from a .tdr file. <a href="#">Reading a Structure on page 57</a>
ILSrc	=<string>	ILS options. <a href="#">Linear Solvers on page 201</a>
ImplicitACSystem		- (g) Construct implicit AC system. <a href="#">AC Analysis in Single-Device Mode on page 152</a>
IncompleteNewton		- (g) Incomplete Newton. <a href="#">Incomplete Newton Algorithm on page 196</a>
(	RhsFactor=<float>	10 Maximum change in RHS to allow old Jacobian to be reused.
	UpdateFactor=<float>)	0.1 Maximum change in update to allow old Jacobian to be reused.
Interrupt	=BreakRequest	Write .tdr or .sav file and terminate actual solve statement after signal INT occurs. <a href="#">Snapshots on page 179</a>
	=PlotRequest	Write .tdr or .sav file and continue simulation after signal INT occurs. <a href="#">Snapshots on page 179</a>

## Appendix G: Command File Overview

### Math

*Table 211 Math{} (Continued)*

Iterations	=<0..>	50 Maximum number of Newton iterations. <a href="#">Convergence and Error Control on page 193</a>
KMC	( <a href="#">Table 222</a> )	KMC metal–insulator–metal (KMC-MIM) transport simulation. <a href="#">Chapter 36 on page 1101</a>
LineSearchDamping	=<(0,1]>	1 Smallest allowed damping coefficient for line search damping. <a href="#">Damped Newton Iterations on page 195</a>
lT_Range	=(<float*2>)	50 5000 K (g) Lower and upper limit for lattice temperature. <a href="#">Numeric Parameters for Temperature Equations on page 256</a>
MetalConductivity		+ Conductivity of metals. The thermal conductivity is simulated in any case. <a href="#">Transport in Metals on page 295</a>
Method	= <a href="#">Table 223</a>	Linear solver for Coupled.
	=Blocked	* Use block decomposition solver for Coupled. <a href="#">Linear Solvers on page 201</a>
(	=Bitlis	Use equation decomposition solver for electrostatic problems. <a href="#">Bitlis Solver on page 918</a>
	Iterations=<int>	200 Number of iterations between successive restarts.
	Restart=<int>	100 Maximum number of iterations.
	Tolerance=<float>)	1e-8 Convergence criterion.

## Appendix G: Command File Overview

### Math

*Table 211 Math{} (Continued)*

MGPreconditioner		Set parameters to control multi-grid algorithm. <a href="#">Bitlis Solver on page 918</a> .
(	MaxCycles=<int>	Controls overall convergence of multi-grid algorithm.
	MaxIterations=<int>	Maximum number of smoother sweeps.
	MaxLevels=<int>	Maximum number of interpolation levels used to build multi-grid hierarchy.
	MinSize=<int>	Minimum size of the coarsest grid.
	RelaxationError=<float>	Controls smoothness of the error.
	SolutionError=<float>)	Controls overall convergence of multi-grid algorithm.
MixAverageBoxMethod		– Use mix average box method algorithm. <a href="#">Box Method Coefficients in the 3D Case on page 1179</a>
MixedModeContactResistance	=<float>	Set a mixed-mode resistance to the circuit node. <a href="#">Ohmic Contacts on page 258</a>
MLDAbox( {	<a href="#">Table 224</a> } ... )	A box within which the interface is confined for the calculation of the MLDA distance function. <a href="#">Modified Local-Density Approximation Model on page 370</a>
MLDAMinDistanceToContact	=<float>	75e-8 μm (g) Minimum distance to contact where MLDA model is switched off. <a href="#">Modified Local-Density Approximation Model on page 370</a>
MobEnormalSignDependence		– (r) Mobility degradation only for carriers that are pulled to the interface by the electric field. <a href="#">Using Mobility Degradation at Interfaces on page 403</a>

## Appendix G: Command File Overview

### Math

*Table 211 Math{} (Continued)*

MobEnormalUniboRegularizationField	=<float>	0 V/cm (r) Regularization field to improve convergence. <a href="#">University of Bologna Surface Mobility Model on page 416</a>
MVMLDAcontrols(	<a href="#">Table 225</a> )	Multivalley MLDA controlling options. <a href="#">Using the MLDA Model on page 373</a>
NaturalBoxMethod		– Use edge-oriented element intersection BM algorithm. <a href="#">Box Method Coefficients in the 3D Case on page 1179</a>
NewHBFfileNames		+ Use new output format. <a href="#">Circuit Currents and Voltages on page 158</a>
NewtonPlot		(g) Convergence monitoring. <a href="#">NewtonPlot on page 209</a>
(	Error	Write the error of all solution variables.
	MinError	Write file only if error decreases.
	NewtonPlotStep=<float>	Upper limit for step size for which to write files.
	Plot	Write everything specified in the Plot section. <a href="#">Device Plots on page 177</a>
	Residual	Write the residuals (right-hand sides) of all equations.
	Update)	Write the updates of all solution variables from the previous step.
NoAutomaticCircuitContact		off (g) Poisson excludes the circuit. <a href="#">Additional Equations Available in Mixed Mode on page 197</a>

## Appendix G: Command File Overview

### Math

*Table 211 Math{} (Continued)*

NonLocal	( <a href="#">Table 226</a> )	(cir) Nonlocal mesh. <a href="#">Nonlocal Meshes on page 202</a>
	<string> ( <a href="#">Table 226</a> )	(g) Named nonlocal mesh.
NonLocalLengthLimit	=<float>	1e-4 cm (g) Limit for nonlocal line length. <a href="#">Nonlocal Meshes on page 202</a>
NonLocalPath	( <a href="#">Table 227</a> )	Dynamic Nonlocal Path Trap-Assisted Tunneling on page 481, Handling Derivatives on page 535
NormalFieldCorrection	=<float>	(r) Normal field correction factor for mobility on interface points. <a href="#">Field Correction on Interface on page 427</a>
NoSRHperPotential		off Omit potential derivatives of SRH recombination rate. <a href="#">Using Field Enhancement on page 478</a>
NoSRHperT		off Omit temperature derivatives of SRH recombination rate. <a href="#">Using Field Enhancement on page 478</a>
NotDamped	=<0..>	1000 (g) Number of iterations in each Newton iteration before Bank–Rose damping is activated. <a href="#">Damped Newton Iterations on page 195</a>
NumberOfAssemblyThreads	=<int>	1 (g) Number of threads for assembly. <a href="#">Parallelization on page 221</a>
NumberOfSolverThreads	=<int>	1 (g) Number of threads for linear solver. <a href="#">Parallelization on page 221</a>
NumberOfThreads	=<int>	1 (g) Number of threads for linear solver and assembly. <a href="#">Parallelization on page 221</a>

## Appendix G: Command File Overview

### Math

*Table 211 Math{} (Continued)*

Numerically	[ (Table 342...)]	off (g) Use numeric derivatives. The optional list restricts the numeric computation to specific Jacobian blocks. <a href="#">Derivatives on page 196</a>
ParallelLicense	(Table 25)	(g) Determine the behavior if insufficient parallel licenses are available.
ParallelToInterfaceInBoundaryLayer		+ (r) Controls the computation of driving forces for mobility and avalanche models along interfaces. <a href="#">Field Correction Close to Interfaces on page 452</a>
(	ExternalBoundary	+ Include the external boundary in the interface for which this feature applies.
	ExternalXPlane	+ Include external x-planes in the interface for which this feature implies.
	ExternalYPlane	+ Include external y-planes in the interface for which this feature implies.
	ExternalZPlane	+ Include external z-planes in the interface for which this feature implies.
	FullLayer	off Apply switch to all elements that touch an interface either by a face, an edge, or a vertex.
	Interface	+ Include semiconductor-insulator region interfaces in the interface for which this feature applies.
	PartialLayer)	on Apply switch only to elements that touch an interface by an edge (2D) or a face (in 3D).

## Appendix G: Command File Overview

### Math

*Table 211 Math{} (Continued)*

ParameterInheritance	=Flatten	* Region parameters inherit materialwise parameter settings. <a href="#">Combining Parameter Specifications on page 78</a>
	=None	Region parameter settings cause materialwise settings to be ignored.
PeriodicBC		(g) Periodic boundary conditions (PBCs). <a href="#">Periodic Boundary Conditions on page 284</a>
	<a href="#">Table 342</a>	Equation for which to apply PBCs. If omitted, apply to all equations (RPBC only).
	Coordinates=<float>*2	μm Coordinate of Direction axis where the PBCs will be applied. Sentaurus Device replaces coordinates outside the device with coordinates at the outer boundary (RPBC only).
	Direction=<0..2>	Direction of periodicity: 0 for x-axis, 1 for y-axis, and 2 for z-axis.
	Factor=<float>	1e8 Tuning parameter $\alpha$ (RPBC only).
	MortarSide=<side>	XMin   YMin   ZMin Mortar side with <side> one of XMin, XMax, YMin, YMax, ZMin, or ZMax (MPBC only).
	Type=<type>) . . . )	RPBC Select PBC mode where <type> is one of RPBC or MPBC.
PlotExplicit		- (g) All Plot statements write as specified in Plot section only.
PlotLoadable		+ (g) All Plot statements write additional information required to load a simulation from a .tdx file.

## Appendix G: Command File Overview

### Math

*Table 211 Math{} (Continued)*

PostProcess	(Transient=<ident> [( <a href="#">Table 364</a> )])	(g) Use PMI <ident> to postprocess data. <a href="#">Postprocessing for Transient Simulations on page 1444</a>
PrintLinearSolver		(g) Print additional information regarding the linear solver being used. <a href="#">Linear Solvers on page 201</a>
RandomizedVariation	<string> ( <a href="#">Table 231</a> )	(g) Use statistical impedance field method. <a href="#">Statistical Impedance Field Method on page 797</a>
RecBoxIntegr	(<float> <int> <int>)	(1e-2 10 1000) Maximum relative deviation of covered volume, maximum number of levels, maximum number of total rectangles per box.
RecomputeQFP		Keep density variables constant, and recompute quasi-Fermi potentials when the electrostatic potential changes and carrier equations are not solved. <a href="#">Introduction to Carrier Transport Models on page 238</a>
RefDens_eEparallel_ElectricField	=<float>	0 cm <sup>-3</sup> (r) Damping parameter for electron high-field mobility and avalanche driving forces. <a href="#">Interpolation of the Eparallel Driving Force on page 452</a> and <a href="#">Interpolation of Avalanche Driving Forces on page 509</a>
RefDens_eEparallel_ElectricField_Aval	=<float>	0 cm <sup>-3</sup> (r) Damping parameter for electron avalanche driving force. <a href="#">Interpolation of Avalanche Driving Forces on page 509</a>
RefDens_eEparallel_ElectricField_HFS	=<float>	0 cm <sup>-3</sup> (r) Damping parameter for the electron high-field mobility driving force. <a href="#">Interpolation of the Eparallel Driving Force on page 452</a>

## Appendix G: Command File Overview

### Math

*Table 211 Math{} (Continued)*

RefDens_eGradQuasiFermi_ElectricField	=<float>	$0 \text{ cm}^{-3}$ (r) Damping parameter for electron high-field mobility and avalanche driving forces. <a href="#">Interpolation of the GradQuasiFermi Driving Force on page 451</a>
RefDens_eGradQuasiFermi_ElectricField_HFS	=<float>	$0 \text{ cm}^{-3}$ (r) Damping parameter for electron high-field mobility driving force. <a href="#">Interpolation of the GradQuasiFermi Driving Force on page 451</a>
RefDens_eGradQuasiFermi_EparallelToInterface_HFS	=<float>	$0 \text{ cm}^{-3}$ (r) Damping parameter for electron high-field mobility driving force. <a href="#">Interpolation of the GradQuasiFermi Driving Force on page 451</a>
RefDens_eGradQuasiFermi_Zero	=<float>	$0 \text{ cm}^{-3}$ (r) Damping parameter for electron high-field mobility driving force, alias for eDrForceRefDens. <a href="#">Interpolation of Driving Forces to Zero Field on page 450</a>
RefDens_Eparallel_ElectricField	=<float>	$0 \text{ cm}^{-3}$ (r) Damping parameter for electron and hole high-field mobility and avalanche driving forces. <a href="#">Interpolation of the Eparallel Driving Force on page 452</a> and <a href="#">Interpolation of Avalanche Driving Forces on page 509</a>
RefDens_Eparallel_ElectricField_Aval	=<float>	$0 \text{ cm}^{-3}$ (r) Damping parameter for electron and hole avalanche driving forces. <a href="#">Interpolation of Avalanche Driving Forces on page 509</a>
RefDens_Eparallel_ElectricField_HFS	=<float>	$0 \text{ cm}^{-3}$ (r) Damping parameter for the electron and hole high-field mobility driving forces. <a href="#">Interpolation of the Eparallel Driving Force on page 452</a>

## Appendix G: Command File Overview

### Math

*Table 211 Math{} (Continued)*

RefDens_GradQuasiFermi_ElectricField	=<float>	$0 \text{ cm}^{-3}$ (r) Damping parameter for electron and hole high-field mobility and avalanche driving forces. <a href="#">Interpolation of the GradQuasiFermi Driving Force on page 451</a>
RefDens_GradQuasiFermi_ElectricField_HFS	=<float>	$0 \text{ cm}^{-3}$ (r) Damping parameter for electron and hole high-field mobility driving forces. <a href="#">Interpolation of the GradQuasiFermi Driving Force on page 451</a>
RefDens_GradQuasiFermi_EparallelToInterface_HFS	=<float>	$0 \text{ cm}^{-3}$ (r) Damping parameter for electron and hole high-field mobility driving forces. <a href="#">Interpolation of the GradQuasiFermi Driving Force on page 451</a>
RefDens_GradQuasiFermi_Zero	=<float>	$0 \text{ cm}^{-3}$ (r) Damping parameter for electron and hole high-field mobility driving forces, alias for DrForceRefDens. <a href="#">Interpolation of Driving Forces to Zero Field on page 450</a>
RefDens_hEparallel_ElectricField	=<float>	$0 \text{ cm}^{-3}$ (r) Damping parameter for hole high-field mobility and avalanche driving forces. <a href="#">Interpolation of the Eparallel Driving Force on page 452</a> and <a href="#">Interpolation of Avalanche Driving Forces on page 509</a>
RefDens_hEparallel_ElectricField_Aval	=<float>	$0 \text{ cm}^{-3}$ (r) Damping parameter for hole avalanche driving force. <a href="#">Interpolation of Avalanche Driving Forces on page 509</a>
RefDens_hEparallel_ElectricField_HFS	=<float>	$0 \text{ cm}^{-3}$ (r) Damping parameter for the hole high-field mobility driving force. <a href="#">Interpolation of the Eparallel Driving Force on page 452</a>

## Appendix G: Command File Overview

### Math

*Table 211 Math{} (Continued)*

RefDens_hGradQuasiFermi_ElectricField	=<float>	$0 \text{ cm}^{-3}$ (r) Damping parameter for hole high-field mobility and avalanche driving forces. <a href="#">Interpolation of the GradQuasiFermi Driving Force on page 451</a>
RefDens_hGradQuasiFermi_ElectricField_HFS	=<float>	$0 \text{ cm}^{-3}$ (r) Damping parameter for hole high-field mobility driving force. <a href="#">Interpolation of the GradQuasiFermi Driving Force on page 451</a>
RefDens_hGradQuasiFermi_EparallelToInterface_HFS	=<float>	$0 \text{ cm}^{-3}$ (r) Damping parameter for hole high-field mobility driving force. <a href="#">Interpolation of the GradQuasiFermi Driving Force on page 451</a>
RefDens_hGradQuasiFermi_Zero	=<float>	$0 \text{ cm}^{-3}$ (r) Damping parameter for hole high-field mobility driving force, alias for hDrForceRefDens. <a href="#">Interpolation of Driving Forces to Zero Field on page 450</a>
RefNa_QuantumPotential	=<float>	$1e25 \text{ cm}^{-3}$ Semiconductor acceptor doping density for the application of the homogeneous Neumann boundary condition on semiconductor–insulator interfaces. <a href="#">Using the Density Gradient Model on page 363</a>
RefNd_QuantumPotential	=<float>	$1e25 \text{ cm}^{-3}$ Semiconductor donor doping density for the application of the homogeneous Neumann boundary condition on semiconductor–insulator interfaces. <a href="#">Using the Density Gradient Model on page 363</a>
RelErrControl		+ Use relative error control. <a href="#">Convergence and Error Control on page 193</a>

## Appendix G: Command File Overview

### Math

*Table 211 Math{} (Continued)*

RelTermMinDensity	=<(0,)>	1e3 cm <sup>-3</sup> (g) Stabilization parameter for temperature relaxation term. <a href="#">Hydrodynamic Model Parameters on page 254</a>
RelTermMinZeroDensity	=<(0,)>	2e8 cm <sup>-3</sup> (g) Stabilization parameter for temperature relaxation term. <a href="#">Hydrodynamic Model Parameters on page 254</a>
RhsAndUpdateConvergence		– Newton converged if both RHS and update are converged. <a href="#">Convergence and Error Control on page 193</a>
RhsFactor	=<float>	1e10 Maximum increase of the $L_2$ -norm of the RHS between Newton iterations. <a href="#">Convergence and Error Control on page 193</a>
RhsMax	=<float>	1e15 Maximum of $L_2$ -norm of the RHS in each Newton iteration (transient simulations). <a href="#">Convergence and Error Control on page 193</a>
RhsMaxQ	=<float>	1e100 Maximum of $L_2$ -norm of the RHS in each Newton iteration (non-transient simulations). <a href="#">Convergence and Error Control on page 193</a>
RhsMin	=<float>	1e-5 Minimum of $L_2$ -norm of the RHS in each Newton iteration. <a href="#">Convergence and Error Control on page 193</a>
RhsMinFactor	=<float>	1e-5 Factor for RhsMin affecting convergence. <a href="#">Convergence and Error Control on page 193</a>
SaveWithinMinStep		- Add numeric guards to the Plot statement, which guarantees that all the specified points are saved. <a href="#">Saving and Plotting During a Quasistationary on page 130</a>

## Appendix G: Command File Overview

### Math

*Table 211 Math{} (Continued)*

SHECutoff	=<[ 0 , )>	5 . 0 (g) Maximum kinetic energy to be plotted in the SHE distribution model. <a href="#">Using Spherical Harmonics Expansion Method on page 857</a>
SHEIterations	=<0 .. >	20 (g) Number of iterations in the SHE distribution model. <a href="#">Using Spherical Harmonics Expansion Method on page 857</a>
SHEMethod	= <a href="#">Table 223</a>	super (g) Linear solver for the SHE distribution model. <a href="#">Using Spherical Harmonics Expansion Method on page 857</a>
SERefinement	=<1 .. >	1 (g) Number of energy grid intervals inside the phonon energy spacing in the SHE distribution model. <a href="#">Using Spherical Harmonics Expansion Method on page 857</a>
SHESOR		+ Use the successive over-relaxation method in the SHE distribution model. <a href="#">Using Spherical Harmonics Expansion Method on page 857</a>
SHESORParameter	=<[ 1 , 2 )>	1 . 1 (g) Successive over-relaxation parameter in the SHE distribution model. <a href="#">Using Spherical Harmonics Expansion Method on page 857</a>
SHETopMargin	=<[ 0 , )>	1 . 0 (g) Top energy margin in the SHE distribution model. <a href="#">Using Spherical Harmonics Expansion Method on page 857</a>
SimpleBandTailIntegration		+ Use abrupt distribution function for band tail density. <a href="#">Band Tails on page 325</a>

## Appendix G: Command File Overview

### Math

*Table 211 Math{} (Continued)*

SimStats		off (g) Write simulation statistics to current plot file. <a href="#">Simulation Statistics for Plotting and Output on page 211</a>
	(DOE_prefix=<string>)	Use the specified prefix for DOE variables.
	WriteDOE)	Write simulation statistics as DOE variables.
Smooth		off Keep mobility and recombination rates from the previous step to obtain better initial conditions for extreme nonlinear iterations.
Spice_gmin	=<float>	1e-12 $\Omega^{-1}$ (g) SPICE minimum conductance. <a href="#">Math Section on page 111</a>
Spice_Temperature	=<float>	300.15 K (g) Temperature of SPICE circuit. <a href="#">Math Section on page 111</a>
SponEmissionIntegration (GaussianQuadrature(Order	=<int>))	Use Gaussian quadrature to integrate the spontaneous emission spectrum of an LED simulation. <a href="#">Accelerating Gain Calculations and LED Simulations on page 1047</a>
StackSize	=<int>	1000000 byte (g) Stack size per thread. <a href="#">Parallelization on page 221</a>
StressLimit	=<float>	1e100 Pa (g) Upper limit on stress values read from files or specified. Values exceeding the limit will be truncated to the limit. <a href="#">Stress Limits on page 939</a>
StressMobilityDependence	=TensorFactor	<a href="#">Stress Tensor Applied to Low-Field Mobility on page 1000</a>

## Appendix G: Command File Overview

### Math

*Table 211 Math{} (Continued)*

StressSG		off (g) Use anisotropic Scharfetter–Gummel approximation for piezo mobility models. <a href="#">Chapter 28 on page 883</a>
SubMethod	= <a href="#">Table 223</a>	(g) Inner linear solver for Blocked method. <a href="#">Linear Solvers on page 201</a>
Surface	<string> ( <a href="#">Table 361...</a> )	(g) Define a surface. <a href="#">Mobility Degradation Components due to Coulomb Scattering on page 419</a> , <a href="#">Random Geometric Fluctuations on page 790</a>
TensorGridAniso		+ (g) Use tensor-grid approximation for anisotropic piezo mobility. <a href="#">Tensor Grid Option on page 999</a>
TensorGridAniso(	Aniso	+ (g) Use tensor-grid approximation for anisotropic models. <a href="#">Chapter 28 on page 883</a>
	Piezo)	+ (g) Same as TensorGridAniso.
ThinLayer(	Mirror=(<mirror>*3))	(g) Mirror planes for simulation system axes. <mirror> is Min, Max, None, or Both. <a href="#">Geometric Parameters of LayerThickness Command on page 381</a>
TransferredElectronEffect2_eMinDerivativePerField	=<float>	-1e100 cm <sup>2</sup> V <sup>-1</sup> s <sup>-1</sup> (r) Lower bound for velocity derivative $\partial v / \partial F_{\text{hfs}}$ (electrons only). <a href="#">Transferred Electron Model 2 on page 442</a>
TransferredElectronEffect2_hMinDerivativePerField	=<float>	-1e100 cm <sup>2</sup> V <sup>-1</sup> s <sup>-1</sup> (r) Lower bound for velocity derivative $\partial v / \partial F_{\text{hfs}}$ (holes only). <a href="#">Transferred Electron Model 2 on page 442</a>

## Appendix G: Command File Overview

### Math

*Table 211 Math{} (Continued)*

TransferredElectronEffect2_MinDerivativePerField	=<float>	-1e100 cm <sup>2</sup> V <sup>-1</sup> s <sup>-1</sup> (r) Lower bound for velocity derivative $\partial v / \partial F_{\text{hfs}}$ (electrons and holes). <a href="#">Transferred Electron Model 2 on page 442</a>
Transient	=BE	(g) Backward Euler method. <a href="#">Backward Euler Method on page 1199</a>
	=TRBDF	* (g) TRBDF method. <a href="#">TRBDF Composite Method on page 1200</a>
TrapDLN	=<int>	13 (g) Levels to approximate trap energy distribution. <a href="#">Energetic and Spatial Distribution of Traps on page 544</a>
TrapDLN(PeakSampling)	=<int>	13 Apply a peak sampling method to Gaussian and exponential distributions in all trap entries. <a href="#">Energetic and Spatial Distribution of Traps on page 544</a>
Traps(	Damping=<[ 0 , )>	10 Damping for traps for the nonlinear Poisson equation. A value of 0 disables damping. <a href="#">Chapter 17 on page 543</a>
	RegionWiseAssembly)	– Apply regionwise assembly for traps.
UpdateIncrease	=<float>	1.e100 (g) Maximum-allowed update error increase factor per Newton step.
UpdateMax	=<float>	1.e100 (g) Maximum-allowed update error in Newton algorithm.
UpdateQFPinTransientPoisson		+ (g) Update quasi-Fermi potentials in Transient, critical for Poisson-only solution. <a href="#">Numeric Control of Transient Analysis on page 139</a>

## Appendix G: Command File Overview

### Math

Table 211    *Math{}* (Continued)

UseSchurSolver		– (g) Replace <code>Blocked</code> by specialized MPBC solver. <a href="#">Specialized Linear Solver for MPBC on page 288</a>
Volume	<string> ( <a href="#">Table 360...</a> )	(g) Define a volume. <a href="#">Random Band Edge Fluctuations on page 794</a>
Wallclock		– (g) Report wallclock times rather than CPU times after each step in the simulation.
WeightedVoronoiBox		off (g) Compute-weighted coefficients and measure. <a href="#">Weighted Box Method Coefficients on page 1185</a>
WithDistResistOnSemi		- Use improved discretization of contacts overlaying the oxide materials for distributed resistance simulations. <a href="#">Resistive Contacts on page 270</a>

## Appendix G: Command File Overview

### Math

**Table 212     AcceptNewtonParameter() *Relaxed Newton Method***

Digits	=<integer>	Number of relative error digits.
Error (	Electron=<float>	Absolute error parameter for electrons.
	eQuantumPotential=<float>	Absolute error parameter for electron quantum potential.
	eTemperature=<float>	Absolute error parameter for electron temperature.
	Hole=<float>	Absolute error parameter for holes.
	hQuantumPotential=<float>	Absolute error parameter for hole quantum potential.
	hTemperature=<float>	Absolute error parameter for hole temperature.
	Poisson=<float>	Absolute error parameter for potential.
ErrRef (	Temperature=<float>)	Absolute error parameter for lattice temperature.
	Electron=<float>	cm <sup>-3</sup> Reference error parameter for electrons.
	eQuantumPotential=<float>	V Reference error parameter for electron quantum potential.
	eTemperature=<float>	K Reference error parameter for electron temperature.
	Hole=<float>	cm <sup>-3</sup> Reference error parameter for holes.
ErrRef (	hQuantumPotential=<float>)	V Reference error parameter for hole quantum potential.
	hTemperature=<float>)	K Reference error parameter for hole temperature.
InvokeAtDivergence		– Allow relaxed convergence to be considered for diverged solutions.
RelErrControl		Use relative error control.

## Appendix G: Command File Overview

### Math

**Table 212** *AcceptNewtonParameter()* *Relaxed Newton Method (Continued)*

RhsAndUpdateConvergence		Require both RHS and update error convergence.
RhsMin	=<float>	Minimum of RHS norm.
UpdateScale	=<float>	Additional factor for the update error.

**Table 213** *BreakCriteria{} Break Criteria: Conditionally Stopping the Simulation*

Current(	absval=<float>	A $\mu\text{m}^{d-3}$ Upper limit for absolute current value.
	contact=<string>	! Name of contact with break criterion.
	DevName=<ident>	Identifier of circuit device name (mixed mode only).
	maxval=<float>	A $\mu\text{m}^{d-3}$ Upper limit for current.
	minval=<float>	A $\mu\text{m}^{d-3}$ Lower limit for current.
	Node=<ident>)	Identifier of circuit node name (mixed mode only).
CurrentDensity(	DevName=<ident>	Identifier of device (mixed mode only).
	maxval=<float>)	A $\text{cm}^{-2}$ (r) Maximum current density.
DevicePower   OuterDevicePower(	absval=<float>	W $\mu\text{m}^{d-3}$ Upper limit for absolute power value.
	DevName=<ident>	Identifier of circuit device name (mixed mode only).
	maxval=<float>	W $\mu\text{m}^{d-3}$ Upper limit for power value.
	minval=<float>)	W $\mu\text{m}^{d-3}$ Lower limit for power value.
ElectricField(	DevName=<ident>	Identifier of circuit device name (mixed mode only).
	maxval=<float>)	V $\text{cm}^{-1}$ (r) Maximum electric field.
InnerDevicePower	as DevicePower	W $\mu\text{m}^{d-3}$ Break criteria based on inner device power.

## Appendix G: Command File Overview

### Math

**Table 213 BreakCriteria{} Break Criteria: Conditionally Stopping the Simulation (Continued)**

Lattice Temperature(	DevName=<ident>	Identifier of circuit device name (mixed mode only).
	maxval=<float>)	K (r) Maximum lattice temperature.
Voltage	as Current	V Voltage break criteria.

**Table 214 ComputeCarrierPath() High-Field Entrance Position and Time**

End(	DopingWell( point=<vector>))	Define end doping well using probe point point. <a href="#">End Condition on page 182</a>
PathReflection(	closeToContact=<float>)	* 0.0 μm Path is reflected at device boundary. Path search stops when distance to a contact is smaller than specified value (in μm).
SaveEPaths		Save electron paths.
(	list=(<int>...)	List of path indices to be plotted.
	offset=<int>	0 Shift first plotted path.
	skip=<int>	0 Number of paths that are omitted for plotting.
	withPlotData)	+ Add interpolated plot variables.
SaveHPaths		Save hole paths.
(	list=(<int>...)	List of path indices to be plotted.
	offset=<int>	0 Shift first plotted path.
	skip=<int>	0 Number of paths that are omitted for plotting.
	withPlotData)	+ Add interpolated plot variables.

## Appendix G: Command File Overview

### Math

**Table 214      *ComputeCarrierPath()* High-Field Entrance Position and Time (Continued)**

Start		<a href="#">Start Condition on page 181</a>
(	Cuboid( corner1=<vector> corner2=<vector> )	Define cuboid using corner points <code>corner1</code> and <code>corner2</code> . <a href="#">Start From Vertex Positions: Cuboid on page 182</a>
	Cuboid( corner1=<vector> corner2=<vector> grid=<vector> )	Define cuboid using corner points <code>corner1</code> and <code>corner2</code> and subgrid using <code>grid</code> . <a href="#">Start From Vertex Positions: Cuboid on page 182</a>
	DopingWell( point=<vector> )	Define doping well using probe point <code>point</code> . <a href="#">Start From Vertex Positions: Doping Well on page 181</a>
	DopingWell( point=<vector> grid=<vector> )	Define doping well using probe point <code>point</code> and subgrid using <code>grid</code> . <a href="#">Start From Vertex Positions: Doping Well on page 181</a>
	Line(point1=<vector> point2=<vector> )	Define line using points <code>point1</code> and <code>point2</code> . <a href="#">Start From Vertex Positions: Line on page 182</a>
	Line(point1=<vector> point2=<vector> grid=<vector> )	Define line using points <code>point1</code> and <code>point2</code> and subgrid using <code>grid</code> . <a href="#">Start From Vertex Positions: Line on page 182</a>
startFrom	= "element"   "vertex"	"vertex" Path search starts from either the vertex position or the element center position.
thresholdField	=<float>	5e5 V/cm Minimum value of multiplication region.

## Appendix G: Command File Overview

### Math

**Table 215    *ComputeIonizationIntegrals()* Approximate Breakdown Analysis**

ComputeAll		All elements are taken into initial element list.
ComputeAtMaxElectricField		Element with maximal electric field is taken into initial element list.
ComputeAtMaxElectricField( delta=<float>)		* 1e4 V/cm All elements where electric field is higher than maximal electric field minus <code>delta</code> are taken into initial element list.
Direction	=<string>	Electricfield Direction field used for the geometric path search.
MinElectricField	=<float>	1 V/cm All elements where electric field is higher than <code>MinElectricField</code> are taken into initial element list.
PathReflection	( <a href="#">Table 228</a> )	Specification of path reflection options. Path is reflected at device boundary.
PathSearchStopDeltaPotential	=<float> <float>	0 V 0 V Electrostatic potential shifts for the path search stop criterion using the electrostatic potential.
PathSearchStopField	=<float>	* 1000 V/cm Threshold value of the absolute value of the electric field where the path search stops.
PeriodicBC	( <a href="#">Table 229</a> )	Specification of periodic boundary conditions.
SaveEIonIntegralPaths	=<int>	Number of paths to be stored based on electron ionization integral. <a href="#">Visualizing Breakdown Paths on page 515</a>
SaveHIonIntegralPaths	=<int>	Number of paths to be stored based on hole ionization integral. <a href="#">Visualizing Breakdown Paths on page 515</a>
Window	( <a href="#">Table 233</a> )	Specification of geometric shapes for spatial restriction of initial element list.
WriteAll		Output information for all computed paths.

## Appendix G: Command File Overview

### Math

*Table 216 eQPBox() Density Gradient Model*

Attenuation_Length	=<float>	1e-3 $\mu\text{m}$ Length over which the gamma ( $\gamma$ ) parameter is effectively attenuated to zero using error function outside the box.
MaxX	=<float>	1e30 $\mu\text{m}$ Maximum x-coordinate for quantum box.
MinX	=<float>	-1e30 $\mu\text{m}$ Minimum x-coordinate for quantum box.
MaxY	=<float>	1e30 $\mu\text{m}$ Maximum y-coordinate for quantum box.
MinY	=<float>	-1e30 $\mu\text{m}$ Minimum y-coordinate for quantum box.
MaxZ	=<float>	1e30 $\mu\text{m}$ Maximum z-coordinate for quantum box.
MinZ	=<float>	-1e30 $\mu\text{m}$ Minimum z-coordinate for quantum box.

*Table 217 EquilibriumSolution() Equilibrium Solution*

Digits	=<float>	Relative error target.
Iterations	=<0..>	50 Maximum number of Newton iterations. <a href="#">Convergence and Error Control on page 193</a>
LineSearchDamping	=<(0,1]>	1 Smallest allowed damping coefficient for line search damping. <a href="#">Damped Newton Iterations on page 195</a>
NotDamped	=<0..>	1000 (g) Number of iterations in each Newton iteration before Bank–Rose damping is activated. <a href="#">Damped Newton Iterations on page 195</a>
RelErrControl		+ Use relative error control. <a href="#">Convergence and Error Control on page 193</a>

*Table 218 Extrapolate() in Math{}, Quasistationary(), Transient() Extrapolation*

Algorithm	=1   2	<a href="#">1 Switch Algorithm on page 135</a> .
Exclude	( <a href="#">Table 342</a> )	<a href="#">Exclude Equations on page 133</a>
LowDensityLimit	=<float>	1e-100 $\text{cm}^{-3}$ Lower limit for carrier density starting from which the appropriate density is extrapolated.
MinStep	=<float>	<a href="#">Minimum Step of Extrapolation on page 133</a>

## Appendix G: Command File Overview

### Math

**Table 218** *Extrapolate() in Math{}, Quasistationary(), Transient() Extrapolation (Continued)*

NumberOfFailures	=<int>	0 <a href="#">Number of Failed Extrapolations on page 134</a>
Order	=<int>	1 Order of extrapolation in Quasistationary or Transient command.

**Table 219** *GrainFieldGenerator() Creating the Grain Field*

AverageGrainSize	=<float>	μ Approximate average grain size in $V_{reg}$ , or $V_{box}$ if padding is used.
AverageGrainSizeVector	= (<ax>, <ay>, <az>)	μ Allow the grain average size to differ in different directions.
GrainLocation	= ((<x1>, <y1>, <z1>), (<x2>, <y2>, <z2>), ...)	μ Set specific locations where grains are placed in the region.
GrainLocationSize	= ((<sx1>, <sy1>, <sz1>), (<sx2>, <sy2>, <sz2>), ...)	μ Set elliptic radii of grains whose locations are specified with GrainLocation. There must be a one-to-one correspondence between values specified with GrainLocation and GrainLocationSize.
MinimumGrainSize	=<float>	μ Set the smallest-allowed distance between grain centers.
NumberOfGrains	=<int>	Number of grain centers located within $V_{reg}$ .
Padding	=<float>	Units of AverageGrainSize. Extend extremities of region by a distance of Padding $\times a_{gi}$ on all sides before randomizing the grain centers.
PoissonDistribution		Use the calculated number of grain centers as the expectation value for a Poisson distribution random number generator.
RandomSeed	=<int>	0 Seed used for the random number generator. Using the same (nonzero) seed in subsequent simulations allows the grain generation to be repeated. The default means a new randomization for each simulation.

## Appendix G: Command File Overview

### Math

*Table 219 GrainFieldGenerator() Creating the Grain Field (Continued)*

SingleVertexBoundary		+ Mark only one vertex of edges that cross a Voronoï boundary with a negative index. To mark both vertices on an edge, specify -SingleVertexBoundary.
TwoDimensionalGrains		Create grains in two dimensions and extrude in the thinnest region dimension.
UniformSpatialDistribution		Distribute grain centers randomly.

*Table 220 HB{} Harmonic Balance*

LogDensity		- Use logarithm of scaled densities as solution variable.
LogTemperature		- Use logarithm of scaled temperatures as solution variable.
MDFT		- Use MDFT mode.
RhsScale	( <a href="#">Table 342</a> )=<float>	Scaling of RHS in Newton (MDFT and SDFT modes).
sdft_update_norm_R		+ Use update error norm (compatible with Version R-2020.09).
SolveSpectrum(	Name=<string>)	Solve spectrum with (mandatory) name.
{	(<int>*n ) . . . }	List of spectrum multi-indices, where <i>n</i> is the number of tones. <a href="#">Solve Spectrum on page 156</a>
UpdateScale	( <a href="#">Table 342</a> )=<float>	Scaling of update in Newton (MDFT and SDFT modes).
ValueMin	( <a href="#">Table 342</a> )=<float>	Lower bound for quantity in time domain (MDFT mode only).
ValueVariation	( <a href="#">Table 342</a> )=<float>	Allowed variation of quantity in time domain (MDFT mode only).

## Appendix G: Command File Overview

### Math

*Table 221 hQPBox() Density Gradient Model*

Attenuation_Length	=<float>	1e-3 μm Length over which the gamma ( $\gamma$ ) parameter is effectively attenuated to zero using error function outside the box.
MaxX	=<float>	1e30 μm Maximum x-coordinate for quantum box.
MinX	=<float>	-1e30 μm Minimum x-coordinate for quantum box.
MaxY	=<float>	1e30 μm Maximum y-coordinate for quantum box.
MinY	=<float>	-1e30 μm Minimum y-coordinate for quantum box.
MaxZ	=<float>	1e30 μm Maximum z-coordinate for quantum box.
MinZ	=<float>	-1e30 μm Minimum z-coordinate for quantum box.

*Table 222 KMC() KMC Simulation Space and Math Settings*

ApproximateBessel		Specify approximate Bessel function for calculation of InelasticElectrodeToTrapTunneling, InelasticTrapToElectrodeTunneling, and TrapToTrapTunneling(MultiPhonon) tunneling models. <a href="#">Specifying the Approximate Bessel Function on page 1106</a>
CellLength	=<float>	0.0005 μm KMC cell size (for TDDB only).
CurrentCompliance(		Use current compliance.
	Contact=<string>	Name of the contact applying current compliance. Default: empty string
	MaxVal=<float>	Maximum value accepted; it must be positive. Default: not applied
	MinVal=<float>)	Minimum value accepted; it must be negative. Default: not applied
HalfCellOffset		Shift the KMC simulation space by a half-cell length relative to device structure (for TDDB only).
IntegrationPoints	=<int>	Set the number of integration points to use with Gauss– Legendre method. <a href="#">Setting the Integration Points on page 1106</a>

## Appendix G: Command File Overview

### Math

**Table 222 KMC() KMC Simulation Space and Math Settings (Continued)**

MaxCurrent	=<float>	0 Use maximum current as criterion to stop TDDB calculation.
MaxCurrentRatio	=<float>	0 Use maximum current ratio as criterion to stop TDDB calculation. The current ratio is defined as $I(t)/\min(I(t'))$ for $t' < t$ .
MaxLocation	=(<float>, <float>, <float>)	μm Define KMC simulation space. <a href="#">KMC Simulation Space on page 1102</a>
MaxTrapNumber	=<int>	0 Use maximum trap number as criterion to stop TDDB calculation.
MaxTrapNumberIncrease	=<int>	0 Each time step allows a maximum number of particles to increase if <code>UseKMCTimeStep</code> is provided.
MIMBackAndForthThreshold	=<int>	2000 Set number of back-and-forth events that occur before rate is adjusted. <a href="#">Adjusting the Calculated Tunneling Rates on page 1106</a>
MIMEEventCountCheckPoint	=<int>	10000 Specify how often the KMC simulator checks for the need to adjust tunneling rates. <a href="#">Adjusting the Calculated Tunneling Rates on page 1106</a>
MinLocation	=(<float>, <float>, <float>)	μm Define KMC simulation space. <a href="#">KMC Simulation Space on page 1102</a>
MoreFrequentRateResets		Reset all MIM tunneling rates after each time step. <a href="#">Resetting the Rate More Frequently on page 1105</a>
RandomSeed	=<int>	Set random seed. <a href="#">Specifying the Random Seed on page 1103</a>
ReferenceElectrode	=<string>	Specify electrode to use as a reference for distances.
ResetElectronFill		Reset the fraction of defects filled with electrons. <a href="#">Resetting the Fraction of Defects Filled With Electrons on page 1104</a>
SkipKMC		Omit KMC simulation completely. <a href="#">Omitting the KMC Simulation Completely on page 1104</a>

## Appendix G: Command File Overview

### Math

**Table 222 KMC() KMC Simulation Space and Math Settings (Continued)**

SkipKMCCurrentCheck	=<float>	Omit further calls to KMC simulator for remainder of Sentaurus Device transient ramp. <a href="#">Omitting KMC Simulations Near Steady State on page 1104</a>
SkipKMCNumber	=<int>	Set number of consecutive Sentaurus Device time-steps before omitting KMC. <a href="#">Omitting KMC Simulations Near Steady State on page 1104</a>
TargetParticle(	Filament{i})	Particles used to determine the time step, which can include Filament1 to Filament3.
Trap2ElecMinDist	=<float>	0.0003 μm Set minimum-allowed defect-to-electrode distance. <a href="#">Setting the Minimum Defect Distances on page 1103</a>
Trap2TrapMinDist	=<float>	0.0003 μm Set minimum-allowed defect-to-defect distance. <a href="#">Setting the Minimum Defect Distances on page 1103</a>
UseAllCrossings		Sample potential across all crossings. <a href="#">Sampling Potential at All Crossings on page 1105</a>
UseKMCTimeStep		– KMC simulator determines the time step for the transient calculation using MaxTrapNumberIncrease when the trap increases sufficiently fast.

**Table 223 Linear solvers (Solvers User Guide)**

ILS		Parallel, iterative linear solver. Customizable, high accuracy, and parallel performance for all problems. Default for 3D structures using set (1).
(	MultipleRHS	– Solve linear systems with multiple right-hand sides (only for AC analysis).
	Set=<int>)	Use ILS options from set <int>.

## Appendix G: Command File Overview

### Math

**Table 223** *Linear solvers (Solvers User Guide) (Continued)*

ParDiSo (		Parallel, supernodal direct solver. High accuracy and parallel performance for small and medium problems.
	IterativeRefinement	– Perform up to two iterative refinement steps to improve the accuracy of the solution.
	MultipleRHS	– Solve linear systems with multiple right-hand sides (only for AC analysis).
	NonsymmetricPermutation	+ Compute an initial nonsymmetric matrix permutation and scaling, which places large matrix entries on the diagonal.
	RecomputeNonsymmetric Permutation)	– Compute a nonsymmetric matrix permutation and scaling before each factorization.
Super		Supernodal direct solver. Best accuracy for small problems, not parallelized. Default for 1D and 2D structures.

**Table 224** *MLDAbox({...}) Modified Local-Density Approximation Model*

MaxX	=<float>	μm Upper x-coordinate of the box.
MaxY	=<float>	μm Upper y-coordinate of the box.
MaxZ	=<float>	μm Upper z-coordinate of the box.
MinX	=<float>	μm Lower x-coordinate of the box.
MinY	=<float>	μm Lower y-coordinate of the box.
MinZ	=<float>	μm Lower z-coordinate of the box.

**Table 225** *MVMLDAcontrols() Using the MLDA Model*

AveDistanceFactor	=<float>	0.05 Factor that controls computation of an averaged distance from a vertex to the interface for the multivalley MLDA model.
Load	=<string>	Name of file from which to load energy-dependent data.
LoadWithInterpolation	=<string>	Name of file from which to load energy-dependent data, possibly obtained for different mesh.

## Appendix G: Command File Overview

### Math

**Table 225** MVMLDAcontrols() *Using the MLDA Model (Continued)*

MaxDoping4Majority	=<float>	1e22 cm <sup>-3</sup> Maximum doping concentration where the multivalley MLDA model is applied to majority carriers.
MaxIntDistance	=<float>	1e-6 cm Distance from the interface up to which the multivalley MLDA model is applied.
Save	=<string>	Name of file where to save energy-dependent data.

**Table 226** Nonlocal() *Nonlocal Meshes*

<a href="#">Table 361</a>		(g) Interface that is part of the reference surface.
Barrier	( <a href="#">Table 360...</a> )	– (g) Regions that form the tunneling barrier.
Digits	=<float>	2 (cgi) Accuracy for nonlocal tunneling currents. <a href="#">Nonlocal Tunneling Parameters on page 830</a>
Direction	=<vector>	(0 0 0) (cgi) If nonzero, suppress the construction of nonlocal mesh lines with a direction more than MaxAngle degrees different from <vector>.
Discretization	=<float>	1e100 cm (cgi) Maximum distance between nonlocal mesh points on a nonlocal line.
Electrode	=<string>	(g) Electrode that is part of the reference surface.
Endpoint		(r) Allow nonlocal lines that end in the region. Default is Endpoint for semiconductors; otherwise, -Endpoint.
	( <a href="#">Table 360...</a> )	(g) Regions where nonlocal lines can or cannot end.
EnergyResolution	=<(0,)>	0.005 eV (cgi) Minimum energy resolution for integrals in computation of nonlocal tunneling current. <a href="#">Nonlocal Tunneling Parameters on page 830</a>
Length	=<float>	cm (cgi) Distance from the interface or contact up to which nonlocal mesh lines are constructed.
MaxAngle	=<float>	180 deg (cgi) Suppress construction of nonlocal mesh lines that enclose an angle of more than <float> degrees with the vector specified by Direction.
Outside		+ (cgi) Allow nonlocal mesh lines to leave the device.

## Appendix G: Command File Overview

### Math

**Table 226 Nonlocal() Nonlocal Meshes (Continued)**

Permeable		+ (r) Allow extension of nonlocal lines (as specified by the Permeation parameter) into or across the region.
	(Table 360...)	+ (g) Regions into or across which nonlocal lines can or cannot be extended.
Permeation	=<float>	0 cm (cgi) Length by which nonlocal mesh lines are extended across the interface or contact.
Refined		+ (r) Autorefine nonlocal lines inside the region.
	(Table 360...)	+ (g) Regions in which nonlocal lines are or are not autorefined.
Transparent		+ (r) Allow nonlocal lines crossing the region.
	(Table 360...)	+ (g) Regions that nonlocal lines can or cannot cross.

**Table 227 NonlocalPath() in Math{}, Quasistationary(), and Transient() Handling Derivatives**

Derivative	=<int>	0: Without derivative computation (default) 1: With derivative computation. <a href="#">Handling Derivatives on page 535</a>
Direction	=<vector>	Frozen tunneling direction.
MaxStep	<float>	(1.0) Higher boundary for step-size control.
MinStep	<float>	(1.0e-6) Lower boundary for step-size control.
N	=<int>	(3) Number of Newton iterations where Jacobian entries are collected.
Postprocessing		– Switch on and off postprocessing mode.
Strategy	=<int>	Strategy for filling of system Jacobian: 1: Collect all 2: Collect last N (default) 3: Collect last N with step size control

## Appendix G: Command File Overview

### Math

Table 228 *PathReflection()* Approximate Breakdown Analysis

CloseToContact	=<float>	0 . 1 Path search stops when distance to a contact is smaller than specified value (in $\mu\text{m}$ ).
ReflectAtMaterial	=<string>	Path is reflected at specified material.
ReflectAtRegion	=<string>	Path is reflected at specified region.

Table 229 *PeriodicBC()* Approximate Breakdown Analysis

Direction	=0   1   2	Direction of periodicity: 0 for x-axis 1 for y-axis 2 for z-axis
Coordinates	=(<float>*2))	$\mu\text{m}$ Coordinate of left and right periodic boundary plane.
MaxReflections	=<int>	5000 Maximum number of periodic path reflections until the path search stops.

Table 230 *RandomField()* Spatial Correlations and Random Fields

AverageGrainSize	=<vector>	$\mu\text{m}$ Average grain size along main axes.
CorrelationFunction	=Exponential	Use exponential correlations.
	=Gaussian	Use Gaussian correlations.
	=Grain	* Use grain-based correlations.
Lambda	=<vector>	$\mu\text{m}$ Correlation length for main axes
MaxInternalPoints	=<int>	2147483647 Maximum-allowed number of points for Fourier transform.
Resolution	=<vector>	(0 . 25 0 . 25 0 . 25) Spatial resolution of exponential and Gaussian randomization.

## Appendix G: Command File Overview

### Math

**Table 231** *RandomizedVariation() in Math{} Statistical Impedance Field Method*

ExtrudeTo3D		– For correlated variations, internally extrude 2D structures to three dimensions.
NumberOfSamples	=<0...>	! Number of samples.
RandomField(	<a href="#">Table 230</a>	(g) Declare random field. <a href="#">Spatial Correlations and Random Fields on page 799</a>
Randomize	=<int>	0 Seed for random number generator.

**Table 232** *Transient time-step control [Numeric Control of Transient Analysis](#)*

CheckTransientError		off Error control of transient integration method.
NoCheckTransientError		on No error control of transient integration method.
TransientDigits	=<float>	3 Relative accuracy for time-step control.
TransientError	( <a href="#">Table 342</a> )=<float>	Absolute error for time-step control.
TransientErrRef	( <a href="#">Table 342</a> )=<float>	Error reference for time-step control.
TrStepRejectionFactor	=<float>	Factor $f_{\text{rej}}$ . <a href="#">Controlling Transient Simulations on page 1201</a>

**Table 233** *Window() [Approximate Breakdown Analysis](#)*

Cuboid(	corner1=<vector>	First corner of cuboid.
	corner2=<vector>)	Diagonally opposite corner of cuboid.
Cylinder(	center1=<vector>	Center of first circular base of cylinder.
	center2=<vector>	Center of second circular base of cylinder.
	radius=<float>)	Radius of cylinder.
Sphere(	center=<vector>	Center of sphere.
	radius=<float>)	Radius of sphere.

## Appendix G: Command File Overview

### NoisePlot

---

## NoisePlot

Table 234    *NoisePlot{}* [Noise Output Data](#)

<a href="#">Table 195</a>	Scalar plot data.
<a href="#">Table 196</a> /Vector	Vector plot data.
AllLNS	All used local noise sources.
AllLNVSd	All used local noise voltage spectral densities.
AllLNvxVsD	All use local noise voltage cross-correlation spectral densities.
GreenFunctions	All used Green's functions and their gradients.

---

## NonLocalPlot

Table 235    *NonLocalPlot{}* [Visualizing Data Defined on Nonlocal Meshes](#)

<a href="#">Table 195</a>		Scalar data.
EigenEnergy(		Eigenenergies. <a href="#">1D Schrödinger Equation on page 351</a>
	Electron[ (Number=<int>) ]	Restrict output to [<int> lowest] electron eigensolutions.
	Hole[ (Number=<int>) ]	Restrict output to [<int> lowest] hole eigensolutions.
OverlapIntegral		Overlap integrals. <a href="#">1D Schrödinger Equation on page 351</a>
WaveFunction(		Wavefunctions. <a href="#">1D Schrödinger Equation on page 351</a>
	Electron[ (Number=<int>) ]	Restrict output to [<int> lowest] electron eigensolutions.
	Hole[ (Number=<int>) ]	Restrict output to [<int> lowest] hole eigensolutions.

---

## OpticalDevice

See [Table 205 on page 1563](#)

## Appendix G: Command File Overview

### Physics

---

## Physics

Table 236 Physics{} Part II, Physics in Sentaurus Device

Active(	Type=QuantumWell)	Activate the localized QW model or the nonlocal QW model to use in conjunction with QWLocal or Schroedinger, respectively.
Affinity	( <ident> [ (Table 364) ] )	(r) Use PMI model <ident> for electron affinity. <a href="#">Electron Affinity on page 1320</a>
AlphaParticle(	<a href="#">Table 282</a> )	(g) Generation by alpha particles. <a href="#">Carrier Generation by Alpha Particles on page 773</a>
AnalyticTEP		(g) Analytic expression for thermoelectric power. <a href="#">Thermoelectric Power on page 1032</a>
Aniso(	<a href="#">Table 285</a> )	Anisotropic properties. <a href="#">Chapter 28 on page 883</a>
AreaFactor	=<float>	1 (g) Multiplier for current and heat flux densities at electrodes and thermodes. <a href="#">Reading a Structure on page 57</a>
BarrierLowering		off (c) Use barrier lowering for Schottky contact. <a href="#">Barrier Lowering at Schottky Contacts on page 265</a>
BreakdownProbability(	<a href="#">Table 241</a> )	Avalanche breakdown probability. <a href="#">Avalanche Breakdown Probability on page 518</a>
ComplexRefractiveIndex(	<a href="#">Table 288</a> )	off (g) Use complex refractive index model. <a href="#">Complex Refractive Index Model on page 694</a>
CondInsulator		(r) Turn an insulator into a conductive insulator. <a href="#">Conductive Insulators on page 300</a>
DefaultParametersFromFile		Initialize default parameters from parameter files instead of using built-in values. <a href="#">Default Parameters on page 84</a>
DeterministicVariation(	<a href="#">Table 290</a> )	(g) Deterministic variations. <a href="#">Deterministic Variations on page 808</a>

## Appendix G: Command File Overview

### Physics

**Table 236 Physics{ Part II, Physics in Sentaurus Device (Continued)**

Dipole(		(i) Use dipole interface model. <a href="#">Dipole Layer on page 230</a>
	Reference=<string>)	! Reference side <string> (either region or material name).
Discontinuity		Create discontinuous interface(s). <a href="#">Discontinuous Interfaces on page 288</a>
DistResist	=<float>	$\Omega\text{cm}^2$ (i) Distributed resistance at metal–semiconductor or metal–metal interfaces. <a href="#">Resistive Interfaces on page 277</a>
	=SchottkyResist	Emulate a Schottky interface. <a href="#">Resistive Interfaces on page 277</a>
eBandTailDOS(	Exponent	off (r) Exponential shape band tails. <a href="#">Band Tails on page 325</a>
	Gaussian	off (r) Gaussian shape band tails. <a href="#">Band Tails on page 325</a>
	GaussianFromOrganic	off (r) Gaussian shape band tails as implemented for organic materials. <a href="#">Band Tails on page 325</a>
	TwoPopulation)	off (r) Two-population model. <a href="#">Band Tails on page 325</a>
eBarrierTunneling	<string> [( <a href="#">Table 242</a> )]	off (g) Nonlocal tunneling from and to conduction band. <a href="#">Nonlocal Tunneling at Interfaces, Contacts, and Junctions on page 826</a>
EffectiveIntrinsic Density	( <a href="#">Table 295</a> )	(r) Band gap and bandgap narrowing. <a href="#">Band Gap and Electron Affinity on page 305</a>
EffectiveMass	(GaussianDOS)	(r) Use simplified Gaussian density-of-states model for organic semiconductors. <a href="#">Gaussian Density-of-States for Organic Semiconductors on page 322</a>
	(<ident> [( <a href="#">Table 364</a> )] )	(r) Use PMI model <ident> for effective mass. <a href="#">Effective Mass on page 1317</a>

## Appendix G: Command File Overview

### Physics

*Table 236 Physics{ Part II, Physics in Sentaurus Device (Continued)*

eMLDA	[ () ]	– (r) MLDA model for electrons. <a href="#">Modified Local-Density Approximation Model on page 370</a>
	( LambdaTemp )	+ (r) Use temperature-dependent $\lambda$ for both electrons and holes.
eMobility(	<a href="#">Table 278</a> )	(r) Electron mobility model. <a href="#">Chapter 15 on page 385</a>
eMultiValley		off (r) Multivalley statistics. <a href="#">Multivalley Band Structure on page 327</a> , <a href="#">Multivalley Band Structure on page 950</a>
eMultiValley(	DensityIntegral	off (r) Numeric density computation. <a href="#">Using Multivalley Band Structure on page 331</a>
	kpDOS	off (r) Two-band $k \cdot p$ models for electron $\Delta_2$ valleys. <a href="#">Using Multivalley Band Structure on page 331</a>
	mCDOS	off (r) Monte Carlo DOS. <a href="#">Using Multivalley Band Structure on page 331</a> , <a href="#">Using the MLDA Model on page 373</a>
	MLDA	off (r) Multivalley MLDA quantization model. <a href="#">Modified Local-Density Approximation Model on page 370</a> , <a href="#">Using Multivalley Band Structure on page 951</a> , <a href="#">Inversion Layer on page 958</a>
	MLDA ( -Nonparabolicity )	off (r) Exclude band nonparabolicity in MLDA model. <a href="#">Using the MLDA Model on page 373</a>
	Nonparabolicity	off (r) Band nonparabolicity. <a href="#">Nonparabolic Band Structure on page 328</a>
	parfile	on (r) With kpDOS, it adds the parameter file valleys. <a href="#">Using Multivalley Band Structure on page 331</a>
	RelativeToDOSMass	off (r) Multivalley effective DOS. <a href="#">Using Multivalley Band Structure on page 331</a>
	ThinLayer )	off (r) Geometric quantization model. <a href="#">Using Multivalley Band Structure on page 331</a> , <a href="#">Using the MLDA Model on page 373</a>

## Appendix G: Command File Overview

### Physics

*Table 236 Physics{ Part II, Physics in Sentaurus Device (Continued)*

EnergyRelaxationTimes	(<ident> [(Table 364)])	Use PMI model <ident> to compute energy relaxation times. <a href="#">Energy Relaxation Times on page 1453</a>
	(constant)	Use constant energy relaxation times.
	(formula)	Use the value of formula in the parameter file.
	(irrational)	Use the ratio of two irrational polynomials. <a href="#">Energy-Dependent Energy Relaxation Time on page 876</a>
eNMP(	Table 297)	(i) Extended nonradiative multiphonon (eNMP) model. <a href="#">Extended Nonradiative Multiphonon Model on page 625</a>
eQCvanDort		off (r) van Dort model for electrons. <a href="#">The van Dort Model on page 349</a>
eQuantumPotential	[(Table 299)]	– (r) Activate electron density gradient quantum correction. <a href="#">Density Gradient Model on page 362</a>
eQuasiFermi	=<float>	V (r) Initial quasi-Fermi potential specification for electrons. <a href="#">Regionwise Specification of Initial Quasi-Fermi Potentials on page 235</a>
eRecVelocity	=<[ 0 , )>	2.573e6 cm s <sup>-1</sup> (i) Electron recombination velocity. <a href="#">Electric Boundary Conditions for Metals on page 296</a>
eSHEDistribution		off (r) Specify the region where the electron SHE distribution is calculated. <a href="#">Using Spherical Harmonics Expansion Method on page 857</a>
	(Table 300)	
eThermionic		(i) Thermionic emission model for electrons. <a href="#">Conductive Insulators on page 300</a> , <a href="#">Thermionic Emission Current on page 870</a>
	(HCI)	(i) Inject hot electrons locally. <a href="#">Destination of Injected Current on page 845</a>
	(Organic_Gaussian)	(i) Thermionic-like Gaussian emission model at organic heterointerfaces for electrons. <a href="#">Gaussian Transport Across Organic Heterointerfaces on page 873</a>

## Appendix G: Command File Overview

### Physics

*Table 236 Physics{ Part II, Physics in Sentaurus Device (Continued)*

ExternalBoltzmannSolver	<string> ()	(g) Connection to external Boltzmann solver. <a href="#">External Boltzmann Solver on page 359</a>
ExternalSchroedinger	<string> ( <a href="#">Table 302</a> )	(g) Connection to external 2D Schrödinger solver. <a href="#">External 2D Schrödinger Solver on page 357</a>
FEPolarization	( <a href="#">Table 156</a> )	Ferroelectric polarization model. <a href="#">Ginzburg–Landau–Khalatnikov Equation on page 910</a>
FEPolarization(	( <a href="#">Table 156</a> )	Landau coefficients. <a href="#">Using the Ginzburg–Landau–Khalatnikov Equation on page 911</a>
	Direction	Direction of the ferroelectric polarization component being solved. <a href="#">Default on page 915</a>
	PMIModel	Landau expansion PMI. <a href="#">Ferroelectrics on page 1413</a>
	SFactor	Dataset name for initial polarization. <a href="#">Using the Ginzburg–Landau–Khalatnikov Equation on page 911</a>
	Tensor)	Anisotropic formulation of the model. <a href="#">Using the Ginzburg–Landau–Khalatnikov Equation on page 911</a>
Fermi		off (g) Fermi statistics. <a href="#">Fermi Statistics on page 233</a>
	(-WithJoyceDixon)	(g) Fermi statistics with old Wuensche approximation for Fermi integrals. <a href="#">Fermi Statistics on page 233</a>
FloatCoef	=<float>	0 (g) Interpolation coefficient for initial guess in floating wells. <a href="#">Initial Guess for Electrostatic Potential and Quasi-Fermi Potentials in Doping Wells on page 234</a>
GateCurrent(	<a href="#">Table 303</a> )	(i) Gate currents (hot-carrier injection, some of the tunneling models).
GaussianDOS_full		(r) Use Gaussian density-of-states model for organic semiconductors. <a href="#">Gaussian Density-of-States for Organic Semiconductors on page 322</a>

## Appendix G: Command File Overview

### Physics

*Table 236 Physics{ Part II, Physics in Sentaurus Device (Continued)*

hBandTailDOS(	Exponent	off (r) Exponential shape band tails. <a href="#">Band Tails on page 325</a>
	Gaussian	off (r) Gaussian shape band tails. <a href="#">Band Tails on page 325</a>
	GaussianFromOrganic	off (r) Gaussian shape band tails as implemented for organic materials. <a href="#">Band Tails on page 325</a>
	TwoPopulation)	off (r) Two-population model. <a href="#">Band Tails on page 325</a>
hBarrierTunneling	<string> [ ( <a href="#">Table 242</a> ) ]	off (g) Nonlocal tunneling from and to valence band. <a href="#">Nonlocal Tunneling at Interfaces, Contacts, and Junctions on page 826</a>
HeatCapacity(	<a href="#">Table 308</a> )	(r) Heat capacity.
HeatPreFactor	=<float>	1. (r) Scaling factor for lattice heat generation. <a href="#">Scaling of Lattice Heat Generation on page 256</a>
HeatSource(	<ident>)	Name of the PMI model.
HeavyIon(	<a href="#">Table 283</a> )	off (g) Generation by heavy ions. <a href="#">Carrier Generation by Heavy Ions on page 775</a>
HeteroInterface		(i) Double points at interfaces. <a href="#">Abrupt and Graded Heterojunctions on page 59</a>
hMLDA(	[ () ]	- (r) MLDA model for holes. <a href="#">Modified Local-Density Approximation Model on page 370</a>
	LambdaTemp )	+ (r) Use temperature-dependent $\lambda$ for both electrons and holes.
hMobility(	<a href="#">Table 278</a> )	(r) Hole mobility model. <a href="#">Chapter 15 on page 385</a>
hMultiValley		off (r) Multivalley statistics. <a href="#">Multivalley Band Structure on page 327</a> , <a href="#">Multivalley Band Structure on page 950</a>

## Appendix G: Command File Overview

### Physics

Table 236 Physics{} Part II, Physics in Sentaurus Device (Continued)

hMultivalley(	DensityIntegral	off (r) Numeric density computation. <a href="#">Using Multivalley Band Structure on page 331</a>
	kpDOS	off (r) Six-band k·p model for hole bands. <a href="#">Multivalley Band Structure on page 327</a>
	mcDOS	off (r) Monte Carlo DOS. <a href="#">Using Multivalley Band Structure on page 331</a> , <a href="#">Using the MLDA Model on page 373</a>
	MLDA	off (r) Multivalley MLDA quantization model. <a href="#">Modified Local-Density Approximation Model on page 370</a> , <a href="#">Using Multivalley Band Structure on page 951</a>
	MLDA ( -Nonparabolicity )	off (r) Exclude band nonparabolicity in MLDA. <a href="#">Using the MLDA Model on page 373</a>
	Nonparabolicity	off (r) Band nonparabolicity. <a href="#">Nonparabolic Band Structure on page 328</a>
	parfile	on (r) With kpDOS, it adds the parameter file valleys. <a href="#">Using Multivalley Band Structure on page 331</a>
	RelativeToDoSMass	off (r) Multivalley effective DOS. <a href="#">Using Multivalley Band Structure on page 331</a>
	ThinLayer )	off (r) Geometric quantization model. <a href="#">Using Multivalley Band Structure on page 331</a> , <a href="#">Using the MLDA Model on page 373</a>
hQCvanDort		off (r) van Dort model for holes. <a href="#">The van Dort Model on page 349</a>
hQuantumPotential	[ ( <a href="#">Table 299</a> ) ]	– (r) Activate hole density gradient quantum correction. <a href="#">Density Gradient Model on page 362</a>
hQuasiFermi	=<float>	V (r) Initial quasi-Fermi potential specification for holes. <a href="#">Regionwise Specification of Initial Quasi-Fermi Potentials on page 235</a>
hRecVelocity	=< [ 0 , ) >	$1.93 \times 10^6$ cms <sup>-1</sup> (i) Hole recombination velocity. <a href="#">Electric Boundary Conditions for Metals on page 296</a>

## Appendix G: Command File Overview

### Physics

**Table 236 Physics{ Part II, Physics in Sentaurus Device (Continued)**

hSHEDistribution		off (r) Specify the region where the hole SHE distribution is calculated. <a href="#">Using Spherical Harmonics Expansion Method on page 857</a>
	( <a href="#">Table 300</a> )	
hThermionic		(i) Thermionic emission model for holes. <a href="#">Conductive Insulators on page 300</a> , <a href="#">Thermionic Emission Current on page 870</a>
	(HCl)	(i) Inject hot holes locally. <a href="#">Destination of Injected Current on page 845</a>
	(Organic_Gaussian)	(i) Thermionic-like Gaussian emission model at organic heterointerfaces for holes. <a href="#">Gaussian Transport Across Organic Heterointerfaces on page 873</a>
Hydrodynamic	[ () ]	(g) Hydrodynamic model for electrons and holes. <a href="#">Hydrodynamic Model for Current Densities on page 241</a> , <a href="#">Hydrodynamic Model for Temperatures on page 252</a>
	(eTemperature)	(g) Hydrodynamic model for electrons only. <a href="#">Hydrodynamic Model for Current Densities on page 241</a> , <a href="#">Hydrodynamic Model for Temperatures on page 252</a>
	(hTemperature)	(g) Hydrodynamic model for holes only. <a href="#">Hydrodynamic Model for Current Densities on page 241</a> , <a href="#">Hydrodynamic Model for Temperatures on page 252</a>
HydrogenDiffusion		off (ri) Hydrogen transport model without reaction or hydrogen type specification. <a href="#">Using MSC-Hydrogen Transport Degradation Model on page 619</a>
	( <a href="#">Table 310</a> )	off (ri) Hydrogen transport model with reaction or hydrogen type specification. <a href="#">Using MSC-Hydrogen Transport Degradation Model on page 619</a>
IncompleteIonization(	<a href="#">Table 312</a> )	Incomplete ionization. <a href="#">Chapter 13 on page 339</a>
KMC_MIM_Transport(	<a href="#">Table 249</a> )	Tunneling processes for KMC-MIM simulations. <a href="#">Tunneling Processes on page 1112</a>

## Appendix G: Command File Overview

### Physics

*Table 236 Physics{} Part II, Physics in Sentaurus Device (Continued)*

LatticeTemperatureLimit	=<float>	K Maximum lattice temperature. <a href="#">Break Criteria: Conditionally Stopping the Simulation on page 119</a>
LayerThickness(		off Thickness extraction command. <a href="#">Extracting Layer Thickness on page 379</a>
	ChordWeight=<float>	0 Weight of chord length. <a href="#">Thickness Extraction on page 382</a>
	MaxFitWeight=<float>	0 <a href="#">Thickness Extraction on page 382</a>
	MinAngle=(<float>*2)	0 0 Angle constraints. <a href="#">Thickness Extraction on page 382</a>
	Thickness=<float>)	μm Explicit thickness value.
LED(	<a href="#">Table 271</a> )	Activate LED framework.
MagneticField	=<vector>	T (g) Magnetic field. <a href="#">Chapter 32 on page 1015</a>
Mechanics(	<a href="#">Table 313</a> )	(g) Mechanical stress solver. <a href="#">Mechanics Solver on page 1008</a>
MetalResistivity(	<ident> [( <a href="#">Table 364</a> )] )	PMI for metal resistivity. <a href="#">Metal Resistivity on page 1431</a>
MetalWorkfunction(	<a href="#">Table 314</a> )	(r) Metal workfunction. <a href="#">Metal Workfunction on page 298</a>
Mobility(	<a href="#">Table 278</a> )	(r) Mobility model. <a href="#">Chapter 15 on page 385</a>
MoleFraction(	<a href="#">Table 316</a> )	Mole fractions. <a href="#">Mole-Fraction Specification on page 65</a>
MSConfigs(	MSConfig ( <a href="#">Table 317</a> ....)....)	(r) Multistate configurations. <a href="#">Specifying Multistate Configurations on page 586</a>
MSDistResist	=<float>	Ωcm <sup>2</sup> (i) Distributed resistance at metal–semiconductor interface. <a href="#">Resistive Contacts on page 270</a>
	=SchottkyResist	Emulate a Schottky interface. <a href="#">Resistive Contacts on page 270</a>

## Appendix G: Command File Overview

### Physics

*Table 236 Physics{ Part II, Physics in Sentaurus Device (Continued)*

MSPeltierHeat		(i) Peltier heat at metal–semiconductor interfaces or semiconductor contacts. <a href="#">Heating at Contacts, Metal–Semiconductor and Conductive Insulator–Semiconductor Interfaces on page 1035</a>
MultiValley		off (r) Multivalley statistics. <a href="#">Multivalley Band Structure on page 327</a> , <a href="#">Multivalley Band Structure on page 950</a>
MultiValley(	DensityIntegral	off (r) Numeric density computation. <a href="#">Using Multivalley Band Structure on page 331</a>
	kpDOS	off (r) $k \cdot p$ model for electrons and holes. <a href="#">Using Multivalley Band Structure on page 331</a>
	MLDA	off (r) Multivalley MLDA quantization model. <a href="#">Modified Local-Density Approximation Model on page 370</a> , <a href="#">Using Multivalley Band Structure on page 951</a> , <a href="#">Inversion Layer on page 958</a>
	MLDA ( -Nonparabolicity )	off (r) Exclude band nonparabolicity in MLDA. <a href="#">Using the MLDA Model on page 373</a>
	Nonparabolicity	off (r) Band nonparabolicity. <a href="#">Nonparabolic Band Structure on page 328</a>
	parfile	on (r) With kpDOS, it adds the parameter file valleys. <a href="#">Using Multivalley Band Structure on page 331</a>
	RelativeToDOSMass )	off (r) Multivalley effective DOS. <a href="#">Using Multivalley Band Structure on page 331</a>
NBTI(	Table 318 )	(i) NBTI degradation model. <a href="#">Two-Stage NBTI Degradation Model on page 621</a>
Noise	[ <string> ] ( Table 319 )	(r) Noise sources. <a href="#">Noise Sources on page 787</a>
OpticalAbsorptionHeat (	ScalingFactor=<float>	1 <a href="#">Optical Absorption Heat on page 1045</a>
	Table 265 )	<a href="#">Optical Absorption Heat on page 1045</a>
Optics(	Table 321 )	<a href="#">Specifying the Type of Optical Generation Computation on page 649</a>

## Appendix G: Command File Overview

### Physics

**Table 236 Physics{ Part II, Physics in Sentaurus Device (Continued)**

Piezo(	<a href="#">Table 323</a> )	(r) Stress and strain models. <a href="#">Using Stress and Strain on page 937</a>
Piezoelectric_Polarization	( <ident> [ <a href="#">Table 364</a> ] )	Use PMI model <ident> to compute piezoelectric polarization. <a href="#">Piezoelectric Polarization on page 1384</a>
	(strain)	Use strain model to compute piezoelectric polarization. <a href="#">Strain Model on page 1001</a>
	(stress)	Use stress model to compute piezoelectric polarization. <a href="#">Stress Model on page 1003</a>
Polarization		off (r) Use ferroelectric model. <a href="#">Chapter 29 on page 905</a>
	(initialvector=<vector>)	(0,0,0) Initial polarization vector for starting the simulation. <a href="#">Using Ferroelectrics on page 905</a>
	(Memory=<2..>)	10 Maximum nesting of minor loops. <a href="#">Using Ferroelectrics on page 905</a>
	(Surface Reconstruction)	off (r) Improved algorithm for calculating the directional polarization. <a href="#">Using Ferroelectrics on page 905</a>
PostTemperature		(g) Use simplified self-heating model. <a href="#">Uniform Self-Heating on page 246</a>
		* Compute dissipated power as integral of Joule heat over entire device.
	(IV_diss)	(g) Compute dissipated power as $IV$ over all electrodes.
	(IV_diss(<string> ...))	(g) Compute dissipated power as $IV$ over user-selected electrodes.
QWLocal(	<a href="#">Table 256</a> )	Localized QW model parameters. <a href="#">Localized Quantum-Well Model on page 1092</a>
Radiation(	<a href="#">Table 284</a> )	Radiation model. <a href="#">Chapter 22 on page 772</a>

## Appendix G: Command File Overview

### Physics

*Table 236 Physics{ Part II, Physics in Sentaurus Device (Continued)*

RandomizedVariation	<string> ( <a href="#">Table 325</a> )	(g) Set sIFM models. <a href="#">Statistical Impedance Field Method on page 797</a>
RayTraceBC(	<a href="#">Table 258</a> )	Physics material or region interface-based definition of boundary conditions. <a href="#">Boundary Condition for Raytracing on page 713</a>
RecGenHeat		(g) Generation–recombination processes act as heat sources. <a href="#">Hydrodynamic Model for Temperatures on page 252</a>
(	OptGenOffset=<float>	0 .5 Divide contribution of optical generation rate into energy gain/loss terms $H_n$ and $H_p$ . <a href="#">Hydrodynamic Model for Temperatures on page 252</a>
	OptGenWavelength=<float>)	μm Wavelength if optical generation is loaded from file. <a href="#">Hydrodynamic Model for Temperatures on page 252</a>
Recombination(	<a href="#">Table 237</a> )	Generation–recombination model. <a href="#">Chapter 16 on page 473</a>
Schottky		off (i) Use Schottky boundary conditions. <a href="#">Electric Boundary Conditions for Metals on page 296</a>
Schroedinger	( <a href="#">Table 326</a> )	(i) Schrödinger solver. <a href="#">1D Schrödinger Equation on page 351</a>
	<string> ( <a href="#">Table 326</a> )	(g) Schrödinger solver on named nonlocal mesh.
SingletExciton		off (ri) Activate region or interface for singlet exciton equation.
(	BarrierType ( <a href="#">Table 287</a> )	(i) Set the barrier type at organic heterointerface. <a href="#">Using the Singlet Exciton Equation on page 293</a>
	FluxBC	off (i) Impose a zero flux boundary condition on specified interface. <a href="#">Using the Singlet Exciton Equation on page 293</a>
	Recombination ( <a href="#">Table 260</a> )	off (r) Switch on generation and recombination terms in singlet exciton equation. <a href="#">Using the Singlet Exciton Equation on page 293</a>

## Appendix G: Command File Overview

### Physics

**Table 236 Physics{ Part II, Physics in Sentaurus Device (Continued)**

TATNonlocalPathNC	=<float>	cm <sup>-3</sup> (i) Room temperature effective density-of-states $N_C$ for electron trap-assisted tunneling from Schottky contacts or Schottky metal–semiconductor interfaces. <a href="#">Using the Dynamic Nonlocal Path Trap-Assisted Tunneling Model on page 483</a>
Temperature	=<float>	300 K (g) Device (lattice) temperature.
TEPower	( <string> )	(gr) Use PMI to compute thermoelectric power. <a href="#">Using Thermoelectric Power on page 1034</a>
	(Analytic)	(gr) Use analytic thermoelectric power. <a href="#">Using Thermoelectric Power on page 1034</a>
ThermalConductivity(	<a href="#">Table 329</a> )	(r) Thermal conductivity.
Thermionic		(i) Thermionic emission model for electrons and holes. <a href="#">Conductive Insulators on page 300</a> , <a href="#">Thermionic Emission Current on page 870</a>
	(HCI)	(i) Inject hot carriers locally. <a href="#">Destination of Injected Current on page 845</a>
	(Organic_Gaussian)	(i) Thermionic-like Gaussian emission model at organic heterointerfaces. <a href="#">Gaussian Transport Across Organic Heterointerfaces on page 873</a>
Thermodynamic		off (g) Thermodynamic transport model. <a href="#">Thermodynamic Model for Current Densities on page 240</a> , <a href="#">Thermodynamic Model for Lattice Temperature on page 249</a>
	( NLLTunnelingRecGen Heat )	(g) Add also the nonlocal line tunneling heating mechanisms to the right-hand side of <a href="#">Equation 72</a> . <a href="#">Thermodynamic Model for Lattice Temperature on page 249</a>
Traps( (	<a href="#">Table 331</a> ) . . . )	Traps. <a href="#">Chapter 17 on page 543</a>

## Appendix G: Command File Overview

### Physics

## Generation and Recombination

Table 237 *Recombination()* [Chapter 16 on page 473](#)

<ident>	[ ( <a href="#">Table 364</a> ) ]	Use PMI model <ident>. <a href="#">Generation–Recombination on page 1266</a>
Auger		off (r) Auger recombination. <a href="#">Auger Recombination Model on page 489</a>
	(WithGeneration)	off (r) Auger recombination and generation.
Avalanche(	<a href="#">Table 238</a> )	off (r) Impact ionization. <a href="#">Avalanche Generation on page 494</a>
Band2Band(	<a href="#">Table 239</a> )	off (r) <a href="#">Band-to-Band Tunneling Models on page 523</a>
CDL(	<a href="#">Table 264</a> )	off (r) Coupled defect level recombination. <a href="#">Coupled Defect Level Recombination on page 487</a>
ConstantCarrier Generation	(value=<float>)	off (r) Constant carrier generation. <a href="#">Constant Carrier Generation Model on page 494</a>
eAvalanche(	<a href="#">Table 238</a> )	off (r) Electron impact ionization. <a href="#">Avalanche Generation on page 494</a>
hAvalanche(	<a href="#">Table 238</a> )	off (r) Hole impact ionization. <a href="#">Avalanche Generation on page 494</a>
intrinsicRichter		off (r) Extended recombination model for silicon, covering Auger and radiative recombination. <a href="#">Intrinsic Recombination Model for Silicon on page 490</a>
Radiative		– (r) Radiative recombination. <a href="#">Radiative Recombination Model on page 488</a>
SRH(	<a href="#">Table 264</a> )	off (r) Shockley–Read–Hall recombination. <a href="#">Shockley–Read–Hall Recombination on page 473</a>
SurfaceSRH		off (i) Interface Shockley–Read–Hall recombination. <a href="#">Surface SRH Recombination Model on page 486</a>
TrapAssistedAuger		off (r) Trap-assisted Auger recombination. <a href="#">Trap-Assisted Auger Recombination Model on page 485</a>

## Appendix G: Command File Overview

### Physics

*Table 238 Avalanche() Chapter 16 on page 473*

<ident> [(Table 364)]	Use PMI model <ident>. <a href="#">Avalanche Generation on page 1259</a>
BandgapDependence	– Include a dependency on the energy bandgap in the avalanche generation models. <a href="#">Using Avalanche Generation on page 495</a>
CarrierTempDrive	Use temperature driving force (default for hydrodynamic simulation). <a href="#">Avalanche Generation With Hydrodynamic Transport on page 510</a>
ElectricField	Use plain electric field driving force. <a href="#">Approximate Breakdown Analysis on page 511</a>
Eparallel	Use current-parallel electric field driving force. <a href="#">Driving Force on page 509</a>
GradQuasiFermi	* Use gradient quasi-Fermi potential driving force. For hydrodynamic simulation, default is CarrierTempDrive. <a href="#">Driving Force on page 509</a>
Hatakeyama	Use Hatakeyama model. <a href="#">Hatakeyama Avalanche Model on page 505</a>
Lackner	Use Lackner model. <a href="#">Lackner Model on page 500</a>
Okuto	Use Okuto–Crowell model. <a href="#">Okuto–Crowell Model on page 499</a>
UniBo	Use University of Bologna model. <a href="#">University of Bologna Impact Ionization Model on page 501</a>
UniBo2	Use new University of Bologna impact ionization model. <a href="#">New University of Bologna Impact Ionization Model on page 502</a>
vanOverstraeten	* Use van Overstraeten model. <a href="#">van Overstraeten – de Man Model on page 497</a>

*Table 239 Band2Band() Band-to-Band Tunneling Models*

DensityCorrection	=Local	Use density correction. <a href="#">Schenk Density Correction on page 527</a>
	=None	* Use plain densities. <a href="#">Schenk Density Correction on page 527</a>
FranzDispersion		– Use Franz dispersion in the direct nonlocal path model. <a href="#">Using the Nonlocal Path Band-to-Band Tunneling Model on page 533</a>
InterfaceReflection		+ Consider interface reflection in the nonlocal path model. <a href="#">Using Band-to-Band Tunneling on page 524</a>

## Appendix G: Command File Overview

### Physics

*Table 239 Band2Band() Band-to-Band Tunneling Models (Continued)*

Model	=E1	Use simple model with $P = 1$ . <a href="#">Simple Band-to-Band Tunneling Models on page 529</a>
	=E1_5	Use simple model with $P = 1.5$ . <a href="#">Simple Band-to-Band Tunneling Models on page 529</a>
	=E2	Use simple model with $P = 2$ . <a href="#">Simple Band-to-Band Tunneling Models on page 529</a>
	=Hurkx	Use Hurkx model. <a href="#">Hurkx Band-to-Band Tunneling Model on page 527</a>
	=modifiedHurkx	Use modified Hurkx model. <a href="#">Modified Hurkx Band-to-Band Tunneling Model on page 528</a>
	=NonlocalPath	Use nonlocal path model. <a href="#">Dynamic Nonlocal Path Band-to-Band Tunneling Model on page 529</a>
	=Schenk	Use Schenk model. <a href="#">Schenk Band-to-Band Tunneling Model on page 526</a>
ParameterSetName	=(<string>...)	Named parameter sets to be used. <a href="#">Using Band-to-Band Tunneling on page 524</a>

*Table 240 BPM() Beam Propagation Method*

Bidirectional(	Error=<float>	! Relative error used as a break criterion for iterative algorithm in bidirectional beam propagation method.
	Iterations=<int>)	! Maximum number of iterations used as a break criterion for iterative algorithm in bidirectional beam propagation method.

## Appendix G: Command File Overview

### Physics

Table 240 *BPM() Beam Propagation Method (Continued)*

Boundary(	GridNodes=<float>*2	! Number of PML boundary grid nodes to be inserted at left and right sides.
	Order=<1..2>	! Order of spatial variation of complex stretching parameter.
	Side=<string>	! "X", "Y", or "Z".
	StretchingParameter Imag=(<float>*2)	! Minimum and maximum values of imaginary part of stretching parameter.
	StretchingParameter Real=(<float>*2)	! Minimum and maximum values of real part of stretching parameter.
	Type="PML"	! Use PML boundary conditions.
	VacuumGridNodes=(<float>*2))	! Number of vacuum grid nodes to be inserted at left and right sides.
Excitation(	CenterGauss=<vector>	$\mu\text{m}$ Gaussian center. Size of vector is $d - 1$ .
	SigmaGauss=<vector>	$\mu\text{m}$ Gaussian half width. Size of vector is $d - 1$ .
	TruncationPositionX=(<float>*2)	Left and right truncation positions of plane wave for x-axis.
	TruncationPositionY=(<float>*2)	Left and right truncation positions of plane wave for y-axis.
	TruncationSlope=<vector>	Plane-wave truncation slope. Size of vector is $d - 1$ .
	Type="Gaussian"	Use Gaussian excitation.
	Type="PlaneWave" )	Use truncated plane wave excitation.
GridNodes	=<vector>	! Number of grid nodes in each spatial dimension.

## Appendix G: Command File Overview

### Physics

**Table 240** *BPM()* Beam Propagation Method (Continued)

ReferenceRefractiveIndex	=<float>	! Value for reference refractive index. <a href="#">General on page 758</a>
	=average	Use average refractive index in propagation plane as reference refractive index.
	=fieldweighted	Use field-weighted refractive index in propagation plane as reference refractive index.
	=maximum	Use maximum refractive index in propagation plane as reference refractive index.
ReferenceRefractiveIndexDelta	=<float>	ReferenceRefractiveIndexDelta is added to ReferenceRefractiveIndex in the calculation. To be used for fine-tuning the numerics. <a href="#">General on page 758</a>

**Table 241** *BreakdownProbability()* Avalanche Breakdown Probability

InterpolatedDiscretization		Use enhanced path search approach.
MinElectricField	=<float>	1.0e5 V/cm Minimum value of multiplication region.

**Table 242** *eBarrierTunneling()* and *hBarrierTunneling()* Nonlocal Tunneling at Interfaces, Contacts, and Junctions

Band2Band	=None	* No band-to-band tunneling.
	=Full	Include band-to-band tunneling with consistent parallel momentum integral with the direct nonlocal path band-to-band tunneling model.
	=Simple	Include band-to-band tunneling with simple parallel momentum integral.
	=UpsideDown	Include band-to-band tunneling with nonphysical parallel momentum integral.
BandGap		– Allow tunneling into the band gap at the interface for which tunneling is specified. Use this option for backward compatibility only.

## Appendix G: Command File Overview

### Physics

**Table 242    *eBarrierTunneling() and hBarrierTunneling() Nonlocal Tunneling at Interfaces, Contacts, and Junctions (Continued)***

BarrierLowering		– Apply a position-dependent barrier lowering for tunneling barrier.
Multivalley		– Use multivalley band structure. <a href="#">Band-to-Band Contributions to Nonlocal Tunneling Current on page 840</a>
PeltierHeat		– Include Peltier heating terms for tunneling carriers. <a href="#">Equation 835 on page 842</a>
Schroedinger		– Schrödinger equation-based model instead of WKB for tunneling probabilities. This option does not work with the TwoBand or Band2Band option. <a href="#">Schrödinger Equation-Based Tunneling Probability on page 836</a>
Transmission		– Use additional interface transmission coefficients. <a href="#">Equation 824 on page 834</a>
TwoBand		– Use two-band dispersion relation. <a href="#">Equation 823 on page 833</a>

**Table 243    *ElectricField() in SRH() and CDL() SRH Field Enhancement***

DensityCorrection	=Local	(r) Use density correction. <a href="#">Density Correction for Schenk and Hurkx Trap-Assisted Tunneling Models on page 481</a>
	=None	* (r) Use local densities.
Lifetime	=Constant	* (r) No field-enhanced lifetime.
	=Hurkx	(r) Use Hurkx model for lifetime enhancement. <a href="#">Hurkx Trap-Assisted Tunneling Model on page 480</a>
	=Schenk	(r) Use Schenk model for lifetime enhancement. <a href="#">Schenk Trap-Assisted Tunneling Model on page 479</a>

**Table 244    *Farfield(...) in Physics{Optics(OpticalSolver(RayTracing(...)))} Far Field and Sensors for Raytracing***

Origin	=Auto	Automatically compute origin as the center of the device. Auto is the default.
	=<vector>	User-specified origin as a vector in micrometers.
Discretization	=<int>	Default is 360 for two dimensions, and 36 for three dimensions.

## Appendix G: Command File Overview

### Physics

**Table 244** *Farfield(...) in Physics{Optics(OpticalSolver(RayTracing(...)))} Far Field and Sensors for Raytracing (Continued)*

ObservationRadius	=<float>	Radius in micrometers. Default is 1e6 μm, that is, 1 m.
Sensor(	<a href="#">Table 262</a> )	Specify a sensor detector.
SensorSweep(	<a href="#">Table 263</a> )	Specify sensor sweep.

**Table 245** *FDTD() Specifying the Optical Solver*

GenerateMesh		Control of tensor mesh generation using Sentaurus Mesh.
(	ForEachWavelength	Generate tensor mesh whenever the wavelength changes compared with the previous FDTD solution.
	Once	Generate tensor mesh once at the beginning of the simulation.
	Wavelength=(<float>*<0..m>)	Specify list of strictly monotonically increasing wavelengths between which the tensor mesh is generated only once.

**Table 246** *FromFile() Loading Solution of Optical Problem From Files and Controlling Interpolation When Loading Optical Generation Profiles*

DatasetName	=AbsorbedPhotonDensity   OpticalGeneration	AbsorbedPhotonDensity Name of dataset to be loaded from file.
GridInterpolation	=Conservative   Simple	Interpolation algorithm to be used if source and destination grid are different.
IdentifyingParameter	=(<string>...)	! Name of identifying parameters or parameter paths.
ImportDomain(	<a href="#">Table 247</a> )	Specify source and destination regions as well as domain truncation for interpolation.
ProfileIndex	=<int>	0 ID of loaded profiles after sorting with respect to leading IdentifyingParameter.
ShiftVector	=<vector>	(0,0,0) Displacement of source grid.

## Appendix G: Command File Overview

### Physics

**Table 246** *FromFile()* Loading Solution of Optical Problem From Files and Controlling Interpolation When Loading Optical Generation Profiles (Continued)

SpectralInterpolation	=Linear   Off   PiecewiseConstant	off Type of interpolation between loaded profiles.
WeightedAPDIntegration layers	=<int>	Default is the number of regions with a limit of 20. However, you can choose a higher number. <a href="#">Accurate Absorbed Photon Density for 1D Optical Solvers on page 691</a>
WeightedAPDIntegration SetBoundaries	=(<float>...)	Specifies the boundaries of layers measured from the illumination window in propagation direction.

**Table 247** *ImportDomain()* Controlling Interpolation When Loading Optical Generation Profiles

DestinationBoxCorner1	=<vector>	(-Inf, -Inf, -Inf) Lower-left corner of box in destination grid used to restrict interpolation domain.
DestinationBoxCorner2	=<vector>	(Inf, Inf, Inf) Upper-right corner of box in destination grid used to restrict interpolation domain.
DestinationRegions	=(<string>...)	Regions in destination grid selected for interpolation.
SourceBoxCorner1	=<vector>	(-Inf, -Inf, -Inf) Lower-left corner of box in source grid used to restrict interpolation domain.
SourceBoxCorner2	=<vector>	(Inf, Inf, Inf) Upper-right corner of box in source grid used to restrict interpolation domain.
SourceRegions	=(<string>...)	Regions in source grid selected for interpolation.

## Appendix G: Command File Overview

### Physics

*Table 248 KMCDefects() Chapter 37 on page 1153*

Diffusion(Particle{i}()		Diffusion event for specified particle.
	Dipole=<float>	Dipole.
	Ea=<float>	Activation energy.
	Frequency=<float>))	Maximum rate of diffusion.
Filament{i}()		Particles defined by keywords Filament1 to Filament3 are conductive particles and are considered in the conductance equation.
	Exclude=<float>	List of excluded particles.
	Name=<string>	Name of the particle.
	ShareCapacity=<float>)	List of particle sharing capacities.
FilamentGrowth(Filament{i}()		Filament growth of Filament{i} from Particle{j}.
	Dipole=<float>	Dipole.
	Ea=<float>	Activation energy.
	Frequency=<float>))	Maximum rate of diffusion.
	Particle{j}))	Specify particle.
FilamentRecession(Filament{i}()		Filament recession of Filament{i} to Particle{j}.
	Dipole=<float>	Dipole.
	Ea=<float>	Activation energy.
	Frequency=<float>))	Maximum rate of diffusion.
	Particle{j}))	Specify particle.

## Appendix G: Command File Overview

### Physics

**Table 248    KMCDefects() Chapter 37 on page 1153 (Continued)**

FilamentSeeds(		Seeds of filament. Filament growth starts at seeds.
	Filament{i})	Filament seeds for <code>Filament{iWW}</code> .
FrenkelPair{i}(		Frenkel pair involved in the generation or recombination process.
	Filament{i}( region=<string>)	Filament in Frenkel pair; region can be specified if at interface.
	Particle{i}( region=<string>)	Particle in Frenkel pair; region can be specified if at interface.
	SameSite)	Two particles are at the same location (for bulk generation or recombination only).
Generation( FrenkelPair{i}(		Generation event for <code>FrenkelPair{i}</code> .
	Dipole=<float>	Dipole.
	Ea=<float>	Activation energy.
	Frequency=<float>))	Maximum rate of diffusion.
Particle{i}(		Definition of particle type. Sentaurus Device supports three different types of particle and filament.
	Exclude=<float>	List of excluded particles.
	Name=<string>	Name of the particle.
	ShareCapacity=<float>)	List of particle sharing capacities.
Recombination( FrenkelPair{i}(		Recombination event for <code>FrenkelPair{i}</code> .
	Dipole=<float>	Dipole.
	Ea=<float>	Activation energy.
	Frequency=<float>))	Maximum rate of diffusion.

## Appendix G: Command File Overview

### Physics

**Table 249 KMC\_MIM\_Transport() Tunneling Processes**

AvgOccupancyInMIMCharge		Use average occupancy associated with a defect for electron charge. <a href="#">Inelastic Electrode-to-Trap and Trap-to-Electrode Tunneling on page 1116</a>
ConductanceEquation		Switched on only when TDDB specified. <a href="#">Time-Dependent Dielectric Breakdown Simulations on page 1128</a>
ConductivePathCurrent		Specify conductive path current (CPC) analysis. <a href="#">Conductive Path Current Analysis on page 1124</a>
ConductiveRadius	=<float>	Influence radius of defects. The conductivity within this radius will have high conductance defined by parameter file.
DeltaTrapPotential		Use a field-independent capture coefficient $c_0$ in calculations. <a href="#">Inelastic Electrode-to-Trap and Trap-to-Electrode Tunneling on page 1116</a>
DirectTunneling		Select direct tunneling. <a href="#">Direct Tunneling on page 1113</a>
ElasticElectrodeToTrapTunneling		Select elastic electrode-to-trap tunneling. <a href="#">Elastic Electrode-to-Trap and Trap-to-Electrode Tunneling on page 1115</a>
ElasticTrapToElectrodeTunneling		Select elastic trap-to-electrode tunneling. <a href="#">Elastic Electrode-to-Trap and Trap-to-Electrode Tunneling on page 1115</a>
ElasticUsePZeroSZero		Use elastic tunneling rates based on inelastic expressions. Option for <a href="#">Elastic Tunneling on page 1116</a>
EnergyDependentMass		Select to account for energy-dependent mass. <a href="#">Energy-Dependent Mass on page 1121</a>
HeatEquation		Switched on only when TDDB specified. <a href="#">Time-Dependent Dielectric Breakdown Simulations on page 1128</a>
ImageChargeBarrierLowering		Select to account for image charge barrier lowering. <a href="#">Image Charge Barrier Lowering on page 1120</a>

## Appendix G: Command File Overview

### Physics

**Table 249 KMC\_MIM\_Transport() Tunneling Processes (Continued)**

IncludeMIMChargeInPoisson		Include defect and electron charge in solution of Poisson equation. <a href="#">Including the MIM Charge in the Poisson Equation on page 1122</a>
InelasticElectrodeToTrapTunneling		Select inelastic electrode-to-trap tunneling. <a href="#">Image Charge Barrier Lowering on page 1120</a>
InelasticIgnorePMinus		Ignore $p < 0$ terms in inelastic tunneling expressions. <a href="#">Options for Inelastic Tunneling on page 1118</a>
InelasticIncludePZero		Include $p = 0$ term in inelastic tunneling expressions. <a href="#">Options for Inelastic Tunneling on page 1118</a>
InelasticPZeroOnly		Use only $p = 0$ term in inelastic tunneling expressions. <a href="#">Options for Inelastic Tunneling on page 1118</a>
InelasticTrapToElectrodeTunneling		Select inelastic trap-to-electrode tunneling. <a href="#">Inelastic Electrode-to-Trap and Trap-to-Electrode Tunneling on page 1116</a>
KmcReram		Trigger KMC calculation.
PooleFrenkelEmission		Select Poole–Frenkel emission. <a href="#">Poole–Frenkel Emission on page 1114</a>
PrintCPCPaths		Print information about paths used in CPC calculation. <a href="#">Conductive Path Current Analysis on page 1124</a>
PrintCPCRates		Print information about rate calculations used in CPC calculation. <a href="#">Conductive Path Current Analysis on page 1124</a>
PrintDefectOccupancy		Print a defect summary, including occupancy and energy to log file and standard output. <a href="#">Occupancy and Energy on page 1136</a>
PrintMIMEventStatistics		Print MIM event statistics to standard output. <a href="#">Event Statistics on page 1137</a>
PrintMIMRates		Print calculated tunneling rates to log file and standard output. <a href="#">Tunneling Rates on page 1138</a>

## Appendix G: Command File Overview

### Physics

**Table 249 KMC\_MIM\_Transport() Tunneling Processes (Continued)**

PrintRECRates		Print tunnel rates calculated by specifying RateEquationCurrent. <a href="#">Rate Equation Current Calculation on page 1123</a>
RateEquationCurrent		Invoke solution of rate equations for a system of defects and selected tunneling models. <a href="#">Rate Equation Current Calculation on page 1123</a>
SensitivityAnalysis		Specify sensitivity analysis. <a href="#">Sensitivity Analysis on page 1125</a>
(	EtRange= ( $<\text{float}>$ , $<\text{float}>$ , $<\text{float}>$ )  PlotType=1   2   3	eV Specify start, end, and step size for defect energy levels.  1 Set type of plot for Sentaurus Visual to generate. Options are: <ul style="list-style-type: none"> <li>• 1: Plot absolute energy versus <math>X_t</math> (flat band).</li> <li>• 2: Plot absolute energy versus <math>X_t</math> (bias included).</li> <li>• 3: Plot <math>E_t</math> versus <math>X_t</math>.</li> </ul>
	Point= ( $<\text{float}>$ , $<\text{float}>$ , $<\text{float}>$ )	(0, 0, 0) $\mu\text{m}$ Specify path for defect locations.
	PrintAll	Print leakage values for all $(X_t, E_t)$ pairs to log file.
	XtRange= ( $<\text{float}>$ , $<\text{float}>$ , $<\text{float}>$ ))	$\mu\text{m}$ Specify start, end, and step size for defect locations.
TDDB		Account for time-dependent dielectric breakdown. <a href="#">Time-Dependent Dielectric Breakdown Simulations on page 1128</a>
TrapToTrapTunneling	[ (Inelastic Phonon)   (MultiPhonon) ]	Select trap-to-trap tunneling model, both elastic and inelastic. Alternatively, select either inelastic phonon or multiphonon trap-to-trap tunneling model. <a href="#">Trap-to-Trap Tunneling on page 1118</a>

## Appendix G: Command File Overview

### Physics

*Table 250 Medium() Transfer Matrix Method*

ExtinctionCoefficient	=<float>	1
Location	=bottom	Position of medium with respect to extracted layer structure.
	=top	! Position of medium with respect to extracted layer structure.
Material	=<string>	Name of material.
RefractiveIndex	=<float>	1

*Table 251 NonlocalPath() in SRH() Dynamic Nonlocal Path Trap-Assisted Tunneling*

Fermi		– Use Fermi statistics.
Lifetime	=Hurkx	* Use Hurkx model for lifetime enhancement.
	=Schenk	Use Schenk model for lifetime enhancement.
TwoBand		– Use Two-band dispersion for the transmission coefficient.

*Table 252 OptBeam((...) Using Optical Beam Absorption Method*

LayerStackExtraction(	ComplexRefractiveIndex Threshold=<float>	0 Choose threshold value for layer creation when using ElementWise extraction mode.
	Mode=RegionWise   ElementWise	RegionWise Specify whether layer stack is created on a per-element or per-region basis.
	Position=(<float>*3)	Specify starting point of extraction line.
	WindowName=<string>	Reference to illumination window in Excitation section.
	WindowPosition=<ident>)	Center Specify starting point of extraction line in terms of a cardinal direction (North, South, East, West, NorthEast, SouthEast, NorthWest, SouthWest).

## Appendix G: Command File Overview

### Physics

**Table 253    *OpticalGeneration{}* Specifying the Type of Optical Generation Computation and Controlling Interpolation When Loading Optical Generation Profiles**

AutomaticUpdate		+ Controls whether optical generation is recomputed if quantities on which it depends are not up-to-date.
ComputeFrom MonochromaticSource (		<a href="#">Optical Generation From Monochromatic Source on page 651</a>
	Scaling=<float>	1
	TimeDependence( <a href="#">Table 267</a> )	
ComputeFromSpectrum (		<a href="#">Illumination Spectrum on page 651</a>
	KeepSpectralData	Keep spectral data (for example, for plotting or to avoid recomputation of optics) when computing the optical generation resulting from an illumination spectrum.
	RefreshEveryTime	Force recomputation of respective optical generation contribution at every occasion.
	Scaling=<float>	1
	Select( <a href="#">Table 261</a> )	Active parameters of multidimensional spectrum file.
	TimeDependence( <a href="#">Table 267</a> )	
QuantumYield	=<float>	<a href="#">1 Quantum Yield Models on page 658</a>
QuantumYield(	EffectiveAbsorption	<a href="#">Quantum Yield Models on page 658</a>
	Factor=<float>	<a href="#">1 Quantum Yield Models on page 658</a>
	pmiModel=<string>	<a href="#">Quantum Yield Models on page 658</a>
	StepFunction( <a href="#">Table 266</a> )	<a href="#">Quantum Yield Models on page 658</a>
	Unity)	<a href="#">Quantum Yield Models on page 658</a>

## Appendix G: Command File Overview

### Physics

**Table 253     *OpticalGeneration{} Specifying the Type of Optical Generation Computation and Controlling Interpolation When Loading Optical Generation Profiles (Continued)***

ReadFromFile (		Loading and Saving Optical Generation From and to Files on page 656
	DatasetName= AbsorbedPhotonDensity   OpticalGeneration	
	TimeDependence( <a href="#">Table 267</a> )	
	Scaling=<float>	1
	RefreshEveryTime	Force recomputation of respective optical generation contribution at every occasion.
	GridInterpolation= Simple   Conservative	Interpolation algorithm to be used if source and destination grid are different.
	ShiftVector=<vector>	(0, 0, 0) Displacement of source grid.
	ImportDomain( <a href="#">Table 247</a> )	Specify source and destination regions as well as domain truncation for interpolation.
Scaling	=<float>	1
SetConstant (		<a href="#">Constant Optical Generation on page 657</a>
	RefreshEveryTime	Force recomputation of respective optical generation contribution at every occasion.
	TimeDependence( <a href="#">Table 267</a> )	
	Value=<float>)	s <sup>-1</sup> cm <sup>-3</sup> Value for optical generation rate.
TimeDependence(	<a href="#">Table 267</a> )	Specification of type of time dependency.

## Appendix G: Command File Overview

### Physics

*Table 254 OpticalTurningPoints() Optical Turning Points*

Dt	=<float>	Limiting time step for range $[t_0 t_3]$ for time-dependence FromFile only. The default value is given by the smallest time interval specified in the OptGenTransientScaling file defined in the File section. For all other types of time dependence, use Dt to set DtRise, DtPlateau, and DtFall to the same value instead of setting each keyword separately. If any of these keywords is set, it overwrites the value of Dt.
DtFall	=<float>	0 s Limiting time step for range $[t_2 t_3]$ . Does not apply to time-dependence FromFile.
DtFallEnd	=<float>	$(t_1 - t_0)/10$ s Limiting time step at $t_3$ .
DtFallStart	=<float>	$(t_1 - t_0)/10$ s Limiting time step at $t_2$ . Does not apply to time-dependence FromFile.
DtPlateau	=<float>	0 s Limiting time step for range $[t_1 t_2]$ . Does not apply to time-dependence FromFile.
DtRise	=<float>	0 s Limiting time step for range $[t_0 t_1]$ . Does not apply to time-dependence FromFile.
DtRiseEnd	=<float>	$(t_1 - t_0)/10$ s Limiting time step at $t_1$ . Does not apply to time-dependence FromFile.
DtRiseStart	=<float>	$(t_1 - t_0)/10$ s Limiting time step at $t_0$ .
MinAmplitude	=<float>	0.01 Signal amplitude used to define time points $t_0$ and $t_3$ for analytic signals with asymptotic tails.

*Table 255 Physics(...){ RayTraceBC(TMM(...)) }*

LayerStructure{	<float> <string>; <float> <string>; ... <float> <string>}	Definition of multilayer structure used for TMM calculation. First column contains thickness of layer in $\mu\text{m}$ . Second column contains material name of layer.
MapOptGenToRegions(	<string> <string> . ...)	Set list of regions to which the lumped optical generation of the TMM BC is mapped.
QuantumEfficiency	=<float>	Specify the quantum efficiency of the TMM BC optical generation.

## Appendix G: Command File Overview

### Physics

*Table 255 Physics(...){ RayTraceBC(TMM(...)) } (Continued)*

ReferenceMaterial	=<string>	Definition of LayerStructure orientation. The topmost layer in the LayerStructure specification is connected to the region with material ReferenceMaterial.
ReferenceRegion	=<string>	Definition of LayerStructure orientation. The topmost layer in the LayerStructure specification is connected to the region with name ReferenceRegion.

*Table 256 QWLocal() Localized Quantum-Well Model*

eDensityCorrection		– Activate electron quantization model in the quantum well. <a href="#">Quantum-Well Quantization Model on page 378</a>
ElectricFieldDep		+ Switch on electric field dependency for the localized QW model.
hDensityCorrection		– Activate hole quantization model in the quantum well. <a href="#">Quantum-Well Quantization Model on page 378</a>
MaxElectricField	=<float>	1e6 V/cm Cutoff value for electric field.
NumberOfCrystalFieldSplitHoleSubbands	=<int>	Set the maximum number of crystal-field split-hole subbands.
NumberOfElectronSubbands	=<int>	Set the maximum number of electron subbands.
NumberOfHeavyHoleSubbands	=<int>	Set the maximum number of heavy-hole subbands.
NumberOfLightHoleSubbands	=<int>	Set the maximum number of light-hole subbands.
NumberOfValenceBands	=<int>	2 Set the number of valence bands.
Polarization	= TE   TM   Mixed	TE Set the polarization used for the computation of the optical transition matrix element. <a href="#">Optical Transition Matrix Element for Wurtzite Crystals on page 1086</a>

## Appendix G: Command File Overview

### Physics

*Table 256 QWLocal() Localized Quantum-Well Model (Continued)*

PolarizationFactor	=<[0,1]>	1 Set the polarization factor used for the computation of the optical transition matrix element in mixed polarization simulations. <a href="#">Optical Transition Matrix Element for Wurtzite Crystals on page 1086</a>
WidthExtraction()	<a href="#">Table 270</a>	Set QW width extraction parameters.

*Table 257 RayDistribution() Distribution Window of Rays*

Dx	=<float>	Specify discretized x-size for Mode=Equidistant.
Dy	=<float>	Specify discretized y-size for Mode=Equidistant.
Mode	=AutoPopulate	Automatically populate rays within the excitation shape.
	=Equidistant	Equidistant distribution of rays within the excitation shape.
	=MonteCarlo	Monte Carlo distribution of rays within the excitation shape.
NumberOfRays	=<int>	Set number of rays for Mode=MonteCarlo or Mode=AutoPopulate.
Scaling	=<float>	Multiply the rays in this window by a scaling factor.
WindowName	=<string>	Specify name of the ray distribution window. No name sets the RayDistribution section as global.

*Table 258 RayTraceBC() in Physics material or region interface-based definition of boundary condition*

Fresnel		Fresnel boundary condition.
PMIModel	=<ident> [ ( <a href="#">Table 364</a> ) ]	Name of the PMI model associated with this BC contact.
Reflectivity	=<[0,1]>	0 Reflectivity.
TMM()	<a href="#">Table 255</a>	Specification of TMM multilayer structure.
Transmittivity	=<[0,1]>	0 Transmittivity.

## Appendix G: Command File Overview

### Physics

**Table 259** *RayTracing(...) in Physics{Optics(OpticalSolver(...))}, unified raytracing interface Raytracing*

CompactMemoryOption		Activate the compact memory model of raytracing.
DepthLimit	=<int>	Stop tracing a ray after passing through more than <int> material boundaries.
ExternalMaterialCRIFile	=<string>	Include a CRI file to define the external media. <a href="#">External Material in Raytracer on page 723</a>
Farfield(	<a href="#">Table 244</a> )	Activate the far field and sensor feature. <a href="#">Far Field and Sensors for Raytracing on page 730</a>
IntensityAbsCoeff	=<float>	Set nonzero absorption coefficient. <a href="#">Computing Optical Intensity on page 726</a>
MinIntensity	=<float>	Stop tracing a ray when its intensity becomes less than <float> times the original intensity. RelativeMinIntensity is an equivalent keyword.
MonteCarlo		Activate Monte Carlo raytracing.
NonSemiconductor Absorption		Include optical generation calculation in nonsemiconductor region or material.
OmitReflectedRays		Discard all reflected rays from raytracing process.
OmitWeakerRays		Discard the weaker ray at a material interface. This is chosen by comparing the reflectivity and transmittivity.
PlotInterfaceFlux		Activate plotting of interface fluxes on all BCs. <a href="#">Plotting Interface Flux on page 728</a>
PolarizationVector	=Random	Default. Automatically choose a random polarization vector that is perpendicular to the starting ray direction.
	=<vector>	Set a fixed polarization vector for the starting ray.
Print		Create the raytree in the output .tdr file.
Print(Skip(<int>))		Create a reduced raytree by omitting every user-defined subtree count.
RayDistribution(	<a href="#">Table 257</a> )	Create a ray distribution to be used in conjunction with the excitation illumination window.

## Appendix G: Command File Overview

### Physics

**Table 259** *RayTracing(...) in Physics{Optics(OpticalSolver(...))}, unified raytracing interface Raytracing (Continued)*

RedistributeStoppedRays		Distribute the total power accumulated at terminated rays back into the raytree.
RetraceCRIchange	=<float>	Fractional change of the complex refractive index to force retracing of rays.
UserWindow(	<a href="#">Table 269</a>	Input a set of starting rays from a file that is specified by the user. <a href="#">User-Defined Window of Rays on page 709</a>
VirtualRegions{	<string> <string> ... }	Specify virtual regions whereby the raytracer will ignore their existence.
WeightedOptical Generation		Switch on weighted method to distribute optical generation from element to vertices. <a href="#">Weighted Interpolation for Raytrace Optical Generation on page 725</a>

**Table 260** *Recombination() in SingletExciton()*

Bimolecular		off (r) Switch on bimolecular recombination in continuity and singlet exciton equations. <a href="#">Singlet Exciton Equation on page 291</a> , <a href="#">Bimolecular Recombination Model on page 538</a>
Dissociation		off (i) Switch on interface exciton dissociation in continuity and singlet exciton equations. <a href="#">Singlet Exciton Equation on page 291</a> , <a href="#">Exciton Dissociation Model on page 539</a>
eQuenching		off (r) Switch on quenching of singlet exciton due to free electrons. <a href="#">Singlet Exciton Equation on page 291</a>
hQuenching		off (r) Switch on quenching of singlet exciton due to free holes. <a href="#">Singlet Exciton Equation on page 291</a>
radiative		off (r) Switch on directly radiative decay of singlet exciton associated with light emission.
trappedradiative		off (r) Switch on trap-assisted radiative decay of singlet exciton associated with light emission.

## Appendix G: Command File Overview

### Physics

*Table 261 Select() Enhanced Spectrum Control*

AllowDuplicates		off (g) Control whether duplicate entries of the spectrum are ignored.
Condition	=<string>	"true" Define a Boolean Tcl expression to select a subset of a multidimensional spectrum.
Parameter	=(<string>...)	Active parameters of multidimensional spectrum file.
Var	=<float>	o Auxiliary parameter that is used to provide more flexibility when specifying a selection condition for the multidimensional spectrum.

*Table 262 Sensor(...) in unified raytracing far field Far Field and Sensors for Raytracing*

Angular(	Phi=(<float>*2)	Define the range of $\phi$ for the angular sensor.
	Theta=(<float>*2))	Define the range of $\theta$ for the angular sensor.
Line(	Corner1=<vector>	Define first point of line sensor.
	Corner2=<vector>	Define second point of line sensor.
	UseNormalFlux)	Compute the projected-to-normal flux.
Name	=<string>	Name of the sensor.
Rectangle(	AxisAligned	Take Corner1 and Corner2 as opposing corners of the rectangle sensor.
	Corner1=<vector>	Define first corner of rectangle sensor.
	Corner2=<vector>	Define second corner of rectangle sensor.
	Corner3=<vector>	Define third corner of rectangle sensor.
	UseNormalFlux)	Compute the projected-to-normal flux.

## Appendix G: Command File Overview

### Physics

**Table 263** *SensorSweep in unified raytracing far field Far Field and Sensors for Raytracing*

Name	=<string>	Specify name of sensor sweep.
Ndivisions	=<int>	Set number of subdivisions for the collection ring.
Phi	=(<float>*2)	Specify range of $\phi$ for the sensor ring.
Theta	=(<float>*2)	Specify range of $\theta$ for the sensor ring.
VaryPhi		Set the sensor sweep as a latitude ring.
VaryTheta		Set the sensor sweep as a longitudinal ring.

**Table 264** *SRH() and CDL() Shockley–Read–Hall Recombination, Coupled Defect Level Recombination*

<ident>	[ ( <a href="#">Table 364</a> ) ]	Use PMI model <ident> to compute lifetimes. <a href="#">Lifetimes on page 1269</a>
DopingDependence		Doping dependence. <a href="#">SRH Doping Dependence on page 474</a>
ElectricField(	<a href="#">Table 243</a> )	(r) Field enhancement. <a href="#">SRH Field Enhancement on page 477</a>
ExpTempDependence		Exponential temperature dependence. <a href="#">SRH Temperature Dependence on page 476</a>
NonlocalPath(	<a href="#">Table 251</a> )	(r) Nonlocal trap-assisted tunneling enhancement. <a href="#">Dynamic Nonlocal Path Trap-Assisted Tunneling on page 481</a>
TempDependence		Temperature dependence. <a href="#">SRH Temperature Dependence on page 476</a>

**Table 265** *StepFunction() Optical Absorption Heat*

Bandgap		Use $E_g - E_{\text{bgn}}$ as the cutoff energy for interband absorption.
EffectiveBandgap		Use $(E_g - E_{\text{bgn}} + 2 \cdot (3/2)kT)$ as the cutoff energy for interband absorption.
Energy	=<float>	eV Use specified value as the cutoff energy for interband absorption.
Wavelength	=<float>	μm Use specified value as the cutoff energy for interband absorption.

## Appendix G: Command File Overview

### Physics

**Table 266** *StepFunction()* [Quantum Yield Models](#)

Bandgap		Use $E_g$ as the cutoff energy for interband absorption.
EffectiveBandgap		Use $E_g - E_{bgn}$ as the cutoff energy for interband absorption.
Energy	=<float>	eV Use specified value as the cutoff energy for interband absorption.
Wavelength	=<float>	μm Use specified value as the cutoff energy for interband absorption.

**Table 267** *TimeDependence()* [Specifying Time Dependency for Transient Simulations](#)

FromFile		off Reads the time dependency as a table from file.
OpticalTurningPoints(	<a href="#">Table 254</a> )	+ (g) Specification of optical turning points. <a href="#">Optical Turning Points on page 667</a> .
Scaling	=<float>	1 Scaling factor for optical generation.
WavePeriods	=<int>	Number of periods of the periodic signal.
WaveTime	=(<float>*2)	s Time interval ( $t_{\min}, t_{\max}$ ) when the optical generation rate is constant.
WaveTPeriod	=<float>	s Period of periodic signal.
WaveTPeriodOffset	=<float>	s Offset of periodic signal (only applies to linear and Gaussian time dependency).
WaveTSigma	=<float>	s Standard deviation $\sigma_t$ of the temporal Gaussian distribution that describes the decay of the optical generation rate outside the time interval WaveTime.
WaveTSlope	=<float>	s <sup>-1</sup> Slope that characterizes the linear decay of the optical generation rate outside the time interval WaveTime.

## Appendix G: Command File Overview

### Physics

**Table 268 TMM() Transfer Matrix Method**

IntensityPattern		(r) Choose type of intensity pattern.
	=Envelope	Compute the envelope of the optical intensity instead of the regular optical intensity.
	=StandingWave	* Compute the regular optical intensity without applying any algorithm that filters out oscillations on the wavelength scale.
LayerStackExtraction(	ComplexRefractive IndexThreshold=<float>	0 Choose threshold value for layer creation when using ElementWise extraction mode.
	Medium( <a href="#">Table 250</a> )	Specification of media surrounding the extracted layer structure.
	Mode=RegionWise   ElementWise	RegionWise Specify whether layer stack is created on a per-element or per-region basis.
	Position=(<float>*3)	Specify starting point of extraction line.
	WindowName=<string>	Reference to illumination window in Excitation section.
	WindowPosition=<ident>	Center Specify starting point of extraction line in terms of a cardinal direction (North, South, East, West, NorthEast, SouthEast, NorthWest, SouthWest).
NodesPerWavelength	=<float>	Number of nodes per wavelength used for computation of optical field.
PropagationDirection	=Perpendicular   Refractive	Refractive Choose interpolation mode for 1D TMM solution.
RoughInterface		+ Flag interfaces as rough, that is, physics of rough interface scattering is applied.
Scattering(	Angular Discretization=<int>	91 Angular discretization of interval $[-\pi/2, \pi/2]$ used in scattering solver.
	MaxNumberOf Iterations=<int>	10 Break criterion for iterative scattering solver.
	Tolerance=<float>)	1e-3 Break criterion for iterative scattering solver.

## Appendix G: Command File Overview

### Physics

**Table 269** *UserWindow in Physics{Optics(OpticalSolver(Raytracing(...)))}, User-Defined Window of Rays*

NumberOfRays	=<int>	Set number of rays in file.
PolarizationVector	=Random	Generate random polarization for the user input rays.
	=ReadFromExcitation	Read the polarization from the Excitation section.
	=ReadFromFile	Read the polarization vectors from the user input ray file.
RaysFromFile	=<string>	Set file name of the user input rays.

**Table 270** *WidthExtraction() Accelerating Gain Calculations and LED Simulations*

ChordWeight	=<float>	Specify chord weight for width extraction. <a href="#">Thickness Extraction on page 382</a>
MinAngle	=(<float>, <float>)	Specify minimum angles for width extraction. <a href="#">Thickness Extraction on page 382</a>
SideMaterial	=("mat1", ..., "matn")	Specify the materials adjoining the QW.
SideRegion	=("regn1", ..., "regn")	Specify the regions adjoining the QW.

## LED

### Note:

LED simulations present unique challenges that require problem-specific model and numerics setups. Contact TCAD Support for advice if you are interested in simulating LEDs (see [Contacting Your Local TCAD Support Team Directly](#)).

**Table 271** *LED() Chapter 34 on page 1038*

Bandstructure(	CrystalType=Zincblende   Wurtzite)	Zincblende Crystal structure of active region. <a href="#">Electronic Band Structure for Wurtzite Crystals on page 1082</a>
Broadening(	<a href="#">Table 272</a> )	Activate gain-broadening models or nonlinear gain saturation model. <a href="#">Gain-Broadening Models on page 1080</a> , <a href="#">Simple Quantum-Well Subband Model on page 1087</a>

## Appendix G: Command File Overview

### Physics

**Table 271 LED() Chapter 34 on page 1038 (Continued)**

Optics(	Table 274)	Optics.
QWExtension	=AutoDetect	Autodetect width of quantum wells. The quantum-well region must be specified by the keyword <code>Active</code> . <a href="#">Radiative Recombination and Gain Coefficients on page 1077</a>
QWTransport		Use ‘three-point’ QW model with thermionic emission. <a href="#">Radiative Recombination and Gain Coefficients on page 1077</a>
SplitOff	=<float>	eV Spin-orbit split-off energy. <a href="#">Strain Effects on page 1090</a>
SponEmissionCoeff	(<ident> [( <a href="#">Table 364</a> )])	Use PMI model <ident> for spontaneous emission. <a href="#">Importing Gain and Spontaneous Emission Data With PMI on page 1096</a>
SponIntegration	(<float>,<int>)	eV Energy integration range and number of discretization intervals for numeric integration (for LED simulations only). <a href="#">Spontaneous Emission Rate and Power on page 1040</a>
SponScaling	=<float>	Scaling factor for matrix element of spontaneous emission. <a href="#">Radiative Recombination and Gain Coefficients on page 1077</a>
StimEmissionCoeff	(<ident> [( <a href="#">Table 364</a> )])	Use PMI model <ident> for stimulated emission. <a href="#">Importing Gain and Spontaneous Emission Data With PMI on page 1096</a>
StimScaling	=<float>	Scaling factor for matrix element of stimulated emission. <a href="#">Radiative Recombination and Gain Coefficients on page 1077</a>
Strain	(RefLatticeConst=<float>)	m Use strain model for quantum well with given reference lattice constant. Strain parameters are input in the parameter file. <a href="#">Strain Effects on page 1090</a>

## Appendix G: Command File Overview

### Physics

**Table 272** *Broadening()* [Gain-Broadening Models](#)

Gamma	=<float>	eV Broadening coefficient $\Gamma$ .
Type	=CosHyper	Use hyperbolic-cosine broadening. <a href="#">Hyperbolic-Cosine Broadening on page 1081</a>
	=Landsberg	Use Landsberg broadening. <a href="#">Landsberg Broadening on page 1081</a>
	=Lorentzian	Use Lorentzian broadening. <a href="#">Landsberg Broadening on page 1081</a>

**Table 273** *ClusterActive() in LED(Optics(RayTrace())) Clustering Active Vertices*

ClusterQuantity	=Nodes	Clustering by recursive grouping of active vertices.
	=OpticalGridElement	Clustering by grouping active vertices in each active optical element.
	=PlaneArea	Clustering by evenly dividing up QW plane area and grouping active vertices in each area element.
NumberOfClusters	=<int>	Specify number of clusters, only for Nodes or PlaneArea clustering.

**Table 274** *Optics() in LED() LED Optics: Raytracing*

RayTrace(	<a href="#">Table 277</a> )	Raytracing.
-----------	-----------------------------	-------------

**Table 275** *OutputLightToolsFarfieldRays() in LED(Optics(RayTrace())) Interfacing Far-Field Rays to LightTools*

Filename	=<string>	Base file name of the LightTools® ray data file to be output.
SaveType	=Ascii	Choose ASCII format for LightTools ray data file.
	=Binary	Choose binary format for LightTools ray data file.
WavelengthDiscretization	=<int>	Number of discretization for the spectrum. The span of the spectrum is determined automatically.

## Appendix G: Command File Overview

### Physics

**Table 276     OutputLightToolsRays() in LED(Optics(RayTrace(Disable())))** *Interfacing LED Starting Rays to LightTools*

IsotropyType	=InBuilt	Use the internal geodesic distribution of starting rays from each active emission vertex cluster.
	=Random	Use a random distribution of starting rays.
	=UserRays	Use the user input set of starting rays defined by the keyword <code>SourceRaysFromFile</code> (see <a href="#">Table 277</a> ).
RaysPerCluster	=<int>	Set number of starting rays in each active cluster.
SaveType	=Ascii	Choose ASCII format for LightTools ray data file.
	=Binary	Choose binary format for LightTools ray data file.
WavelengthDiscretization	=<int>	Number of discretization for the spectrum. The span of the spectrum is determined automatically.

**Table 277     RayTrace() in LED(Optics())** *LED Optics: Raytracing*

ClusterActive(	<a href="#">Table 273</a> )	Activate the clustering option. <a href="#">Clustering Active Vertices on page 1053</a>
CompactMemoryOption		Activate the compact memory model for LED raytracing. Will not work with the full photon-recycling model.
Coordinates	=Cartesian	*
	=Cylindrical	
DebugLEDRadiation	(<string> <float1> <float2> <float3>)	off Trace the origin of the output rays that are within the angles [<float1>, <float2>] (in degrees) and of minimum intensity specified by the <float3> parameter. In two dimensions, the angle is defined in the regular polar coordinates. In three dimensions, the angles are defined from the z-axis, as in $\theta$ in regular spherical coordinates. The results are output to the file specified by <string>.
DepthLimit	=<int>	5 Stop tracing the ray after passing through more than <int> material boundaries.
Disable		off Disable raytracing but still run LED simulation.

## Appendix G: Command File Overview

### Physics

**Table 277 RayTrace() in LED(Optics()) *LED Optics: Raytracing* (Continued)**

Disable(OutputLightTools Rays(	<a href="#">Table 276</a> ) )	Output starting rays from active vertices to a LightTools ray data file. <a href="#">Interfacing LED Starting Rays to LightTools on page 1057</a>
EmissionType	(Anisotropic( Sine(<float>*3)C osine(<float>*3) ))	
	(Isotropic)	*
ExcludeHorizontalSource	(<float>)	off Omit the source rays that are emitted within the horizontal angular zone specified by the <float> parameter (in degrees).
LEDRadiationPara	(<float>, <int>)	μm Observation radius and discretization of the observation circle or sphere.
LEDSpectrum	(<float>*2 <int>)	eV Starting and ending energy range, and number of subdivisions in that energy range.
MinIntensity	=<float>	1e-5 Stop tracing the ray when the intensity of a ray becomes less than <float> times the original intensity.
MonteCarlo		Activate Monte Carlo raytracing.
MoveBoundaryStartRays	=<float>	0 nm Shift the starting ray position at device boundary inwards. Recommended values are 1 to 5 nm.
NonActiveAbsorptionOff		Do not add nonactive region absorption as generation rate to continuity equation. <a href="#">Nonactive Region Absorption (Photon Recycling) on page 1068</a>
ObservationCenter	=<vector>	Fixed observation center for LED radiation. By default, center of device.
OmitReflectedRays		Discard all reflected rays in raytracing process.
OmitWeakerRays		Discard the weaker ray at a material interface. This is decided by comparing the reflectivity and transmittivity.
OptGenScaling	=<float>	Set a multiplication factor to the nonactive optical generation computed.

## Appendix G: Command File Overview

### Physics

**Table 277 RayTrace() in LED(Optics()) *LED Optics: Raytracing* (Continued)**

OutputLightToolsFarfieldRays()	<a href="#">Table 275</a> )	Output far-field rays with embedded spectrum information into a LightTools ray data file. <a href="#">Interfacing Far-Field Rays to LightTools on page 1067</a>
PolarizationVector	=Random	Default. Automatically choose a random polarization vector that is perpendicular to the starting ray direction.
	=<vector>	Set a user-defined polarization vector.
Print		off Output all rays to the output .tdr file.
	(ActiveVertex (<int>))	off Output only ray paths emitted from this active vertex.
	(Skip(<int>))	1 Output every other <int> ray to the output .tdr file.
PrintRayInfo	(<string>)	off Print all indices and positions of starting rays into the file specified by <string>.
PrintSourceVertices	(<string>)	off Print the index and coordinates of all active vertices into the file specified by <string>.
ProgressMarkers	=<int>	5 Completion meter for raytracing.
RaysPerVertex	=<int>	10 Number of rays starting from each active source vertex. For 3D, the number of starting rays are constrained by 6, 18, 68, and so on. The number in the sequence is chosen such that RaysPerVertex is slightly larger or equal to it.
RaysRandomOffset		Randomize the angular shift of the starting rays.
	(RandomSeed=<int>)	Set a fixed random seed so that repeated simulations will yield the exact pseudorandom results.
RedistributeStoppedRays		Distribute the total accumulated power in terminated rays back into the raytree.
RetraceCRIchange	=<float>	Fractional change of the complex refractive index to force retracing of rays.

## Appendix G: Command File Overview

### Physics

Table 277 *RayTrace() in LED(Optics())* [LED Optics: Raytracing \(Continued\)](#)

SourceRaysFromFile(	<string> )	Import a set of starting ray directions from the file name specified by <string>. The number of imported rays are specified by <code>RaysPerVertex</code> .
Staggered3DFarfieldGrid		Use the staggered 3D far-field collection sphere. <a href="#">Staggered 3D Grid LED Radiation Pattern on page 1063</a>
TraceSource	( )	Retrace the source of the output rays to produce a map of the source regions that give the strongest ray output.
TurnOffTreeNodeCount		Disable counting the total number of nodes in the raytree.
Wavelength	=<float>	nm Wavelength.
	=AutoPeak	Take wavelength at the peak of the spontaneous emission rate spectrum. <a href="#">LED Wavelength on page 1045</a>
	=AutoPeakPower	Take wavelength at the peak of the spontaneous emission power spectrum.
	=Effective	Wavelength computed such that total power = total rate × effective photon energy.

## Appendix G: Command File Overview

### Physics

## Mobility

Table 278 *Mobility(), eMobility(), hMobility()* [Chapter 15 on page 385](#)

BalMob(		off Use ballistic mobility model. <a href="#">Ballistic Mobility Model on page 458</a>
	Fermi	off Use Fermi–Dirac correction in the ballistic mobility. <a href="#">Using the Ballistic Mobility Model on page 460</a>
	Frensley	off Use Frenstley rule for final mobility. <a href="#">Using the Ballistic Mobility Model on page 460</a>
	KineticEnergy	off Use kinetic ballistic mobility model. <a href="#">Using the Ballistic Mobility Model on page 460</a>
	Lch=<float>	$10^7$ nm Use explicit channel length in simple ballistic mobility. <a href="#">Using the Ballistic Mobility Model on page 460</a>
	TempDep )	off Use temperature dependency in simple ballistic mobility. <a href="#">Using the Ballistic Mobility Model on page 460</a>
BandTailHighField Saturation(	<a href="#">Table 279</a> )	off Use band tail mobility. <a href="#">Band Tail Mobility on page 454</a>
BTCSMob(	Nonlocal	off Account for scattering by remote traps. <a href="#">General Model for Mobility Degradation by Traps in the Bulk and at Semiconductor Interfaces on page 462</a>
	NonlocalLengthLimit=<float> )	0 cm Distance up to which to account for remote traps, when Nonlocal option is active.
CarrierCarrier Scattering	(BrooksHerring)	off Use Brooks–Herring carrier–carrier scattering model. <a href="#">Carrier–Carrier Scattering on page 396</a>
	(ConwellWeisskopf)	off Use Conwell–Weisskopf carrier–carrier scattering model. <a href="#">Carrier–Carrier Scattering on page 396</a>
CNTMob(	Lch=<float> )	off Use CNT mobility model. <a href="#">Carbon Nanotube Mobility Model on page 433</a>

## Appendix G: Command File Overview

### Physics

**Table 278    Mobility(), eMobility(), hMobility() [Chapter 15 on page 385 \(Continued\)](#)**

ConstantMobility		+ Use constant mobility if neither PhuMob nor DopingDependence is specified. <a href="#">Mobility due to Phonon Scattering on page 386</a>
Diffusivity(	<a href="#">Table 279</a> )	off Use non-Einstein diffusivity. <a href="#">Non-Einstein Diffusivity on page 453</a>
DopingDependence(	<ident> [ ( <a href="#">Table 364</a> ) ]	off PMI model <ident>. <a href="#">Doping-Dependent Mobility on page 1279</a>
	Arora	off Use Arora doping-dependent mobility model. <a href="#">Doping-Dependent Mobility Degradation on page 387</a>
	BalMob( [ Lch=<float>] )	off Use low-field ballistic mobility model. <a href="#">Low-Field Ballistic Mobility Models on page 394</a>
	Masetti	off Use Masetti doping-dependent mobility model. <a href="#">Doping-Dependent Mobility Degradation on page 387</a>
	PhuMob [ see PhuMob options in <a href="#">Table 278</a> ]	off Use Philips unified mobility model. <a href="#">Philips Unified Mobility Model on page 398</a>
	PhuMob2	off Use an alternative Philips model. <a href="#">Using an Alternative Philips Unified Mobility Model on page 399</a>
	PMIModel( <a href="#">Table 324</a> )	off Use PMI for MSC-dependent mobility. <a href="#">Multistate Configuration-Dependent Apparent Band-Edge Shift on page 1323</a>
	UniBo )	off Use University of Bologna doping-dependent mobility model. <a href="#">Doping-Dependent Mobility Degradation on page 387</a>
eBandTailHighFieldSaturation(	<a href="#">Table 279</a> )	off Use band tail mobility for electrons. <a href="#">Band Tail Mobility on page 454</a>
eDiffusivity(	<a href="#">Table 279</a> )	off Use non-Einstein electron diffusivity. <a href="#">Non-Einstein Diffusivity on page 453</a>
eHighFieldSaturation(	<a href="#">Table 279</a> )	off Electron high-field saturation. <a href="#">High-Field Saturation Models on page 438</a>

## Appendix G: Command File Overview

### Physics

**Table 278** *Mobility(), eMobility(), hMobility()* [Chapter 15 on page 385 \(Continued\)](#)

Enormal(	<ident> [(Table 364)]	off PMI model <ident>. <a href="#">Mobility Degradation at Interfaces on page 1300</a>
	Coulomb2D	off ‘Two-dimensional’ ionized impurity mobility degradation. <a href="#">Mobility Degradation Components due to Coulomb Scattering on page 419</a>
	IALMob( <a href="#">Table 280</a> )	off Inversion and accumulation layer mobility model. <a href="#">Mobility Degradation at Interfaces on page 403</a>
	InterfaceCharge [(SurfaceName=<string>)]	off Negative and positive interface charge mobility degradation. <a href="#">Mobility Degradation Components due to Coulomb Scattering on page 419</a>
	Lombardi( <a href="#">Table 281</a> )	off Enhanced Lombardi model. <a href="#">Mobility Degradation at Interfaces on page 403</a>
	Lombardi_highk	off Enhanced Lombardi model with high-k degradation. <a href="#">Mobility Degradation at Interfaces on page 403</a>
	NegInterfaceCharge [(SurfaceName=<string>)]	off Negative interface charge mobility degradation. <a href="#">Mobility Degradation Components due to Coulomb Scattering on page 419</a>
	PosInterfaceCharge [(SurfaceName=<string>)]	off Positive interface charge mobility degradation. <a href="#">Mobility Degradation Components due to Coulomb Scattering on page 419</a>
	RCS	off Remote Coulomb scattering mobility degradation. <a href="#">Remote Coulomb Scattering Model on page 423</a>
	RPS	off Remote phonon scattering mobility degradation. <a href="#">Remote Phonon Scattering Model on page 425</a>
UniBo)		off University of Bologna surface mobility model. <a href="#">Mobility Degradation at Interfaces on page 403</a>
hBandTailHighField Saturation(	<a href="#">Table 279</a> )	off Use band tail mobility for electrons. <a href="#">Band Tail Mobility on page 454</a>

## Appendix G: Command File Overview

### Physics

**Table 278** *Mobility(), eMobility(), hMobility()* [Chapter 15 on page 385 \(Continued\)](#)

hDiffusivity(	<a href="#">Table 279</a> )	off Use non-Einstein hole diffusivity. <a href="#">Non-Einstein Diffusivity on page 453</a>
hHighFieldSaturation(	<a href="#">Table 279</a> )	off Hole high-field saturation. <a href="#">High-Field Saturation Models on page 438</a>
HighFieldSaturation(	<a href="#">Table 279</a> )	off High-field saturation. <a href="#">High-Field Saturation Models on page 438</a>
IncompleteIonization		off Incomplete ionization-dependent mobility. <a href="#">Incomplete Ionization–Dependent Mobility Models on page 463</a>
PhuMob (		off Philips unified mobility model. <a href="#">Philips Unified Mobility Model on page 398</a>
	Arsenic	* Use arsenic parameters. <a href="#">Table 58 on page 402</a>
	Phosphorus)	Use phosphorus parameters. <a href="#">Table 58 on page 402</a>
ThinLayer (		off Thin-layer mobility model. <a href="#">Thin-Layer Mobility Model on page 427</a>
	ChordWeight=<float>	0 Weight of chord length. <a href="#">Thickness Extraction on page 382</a>
	IALMob( <a href="#">Table 280</a> )	Use the inversion and accumulation layer mobility model with the thin-layer mobility model. <a href="#">Thin-Layer Mobility Model on page 427</a>
	Lombardi( <a href="#">Table 281</a> )	* Use the enhanced Lombardi model with the thin-layer mobility model. <a href="#">Thin-Layer Mobility Model on page 427</a>
	MinAngle=(<float>*2)	0 0 Angle constraints. <a href="#">Thickness Extraction on page 382</a>
	Thickness=<float>)	μm Explicit thickness value.

## Appendix G: Command File Overview

### Physics

**Table 278** *Mobility(), eMobility(), hMobility()* [Chapter 15 on page 385 \(Continued\)](#)

ToCurrentEnormal	as Enormal	off Dependence on electric field normal to current. <a href="#">Mobility Degradation at Interfaces on page 403</a>
Tunneling		– Tunneling correction to mobility. <a href="#">Equation 239 on page 363</a>

**Table 279** *HighFieldSaturation(), Diffusivity()* [High-Field Saturation Models](#)

<ident>	[ <a href="#">(Table 364)</a> ]	PMI model <ident>. <a href="#">High-Field Saturation on page 1287</a>
AutoOrientation		– Use a parameter set based on the orientation of the nearest interface. <a href="#">Auto-Orientation for the High-Field Saturation Models on page 440</a>
CarrierTempDrive		Temperature driving force (default for hydrodynamic simulation). <a href="#">Hydrodynamic Driving Force on page 450</a>
CarrierTempDriveBasic		Basic temperature driving force. <a href="#">Basic Model on page 444</a>
CarrierTempDriveME		Meinerzhagen–Engl temperature driving force. <a href="#">Meinerzhagen–Engl Model on page 445</a>
CarrierTempDrivePolynomial		Energy-dependent mobility model using an irrational polynomial. <a href="#">Energy-Dependent Mobility on page 878</a>
CarrierTempDriveSpline		Energy-dependent mobility model using spline interpolation. <a href="#">Spline Interpolation on page 879</a>
CaugheyThomas		* Use Canali–Hänsch model. <a href="#">Extended Canali Model on page 440</a>
ElectricField		Electric field. <a href="#">Electric Field on page 450</a>
Eparallel		Use electric field parallel to current as driving force. <a href="#">Electric Field Parallel to the Current on page 447</a>
EparallelToInterface		Use electric field parallel to the closest semiconductor–insulator interface as driving force. <a href="#">Electric Field Parallel to the Interface on page 448</a>

## Appendix G: Command File Overview

### Physics

**Table 279     *HighFieldSaturation(), Diffusivity()* High-Field Saturation Models (Continued)**

GradQuasiFermi		* Use gradient of quasi-Fermi potential as driving force. For hydrodynamic simulations, default is CarrierTempDrive. <a href="#">Gradient of Quasi-Fermi Potential on page 448</a>
ParameterSetName	=<string>	Name of parameter set to use. <a href="#">Named Parameter Sets for the High-Field Saturation Models on page 440</a>
PFMob		Use Poole–Frenkel mobility model. <a href="#">Poole–Frenkel Mobility on page 455</a>
PMIModel(	<a href="#">Table 324</a> )	PMI model dependent on two fields. <a href="#">High-Field Saturation With Two Driving Forces on page 1295</a>
TransferredElectronEffect		Use transferred electron model. <a href="#">Transferred Electron Model on page 442</a>
TransferredElectronEffect2		Use an alternative transferred electron model. <a href="#">Transferred Electron Model 2 on page 442</a>
VRHMob		Use variable range hopping mobility model. <a href="#">Variable Range Hopping Transport Mobility on page 456</a>

**Table 280     *IALMob()* Mobility Degradation at Interfaces**

AutoOrientation		– Use a parameter set based on the orientation of the nearest interface. <a href="#">Auto-Orientation for the IALMob Model on page 416</a>
ClusteringEverywhere		– Use clustering formulas for all occurrences of $N_A$ and $N_D$ . <a href="#">Inversion and Accumulation Layer Mobility Model on page 407</a>
FullPhuMob		+ Use the full PhuMob mobility expressions. <a href="#">Inversion and Accumulation Layer Mobility Model on page 407</a>
ParameterSetName	=<string>	Name of <code>IALMob</code> parameter set. <a href="#">Named Parameter Sets for the IALMob Model on page 415</a>
PhononCombination	=<0..2>	<sup>1</sup> Selects how 2D and 3D phonon mobility are combined. <a href="#">Inversion and Accumulation Layer Mobility Model on page 407</a>

## Appendix G: Command File Overview

### Physics

Table 281 Lombardi() *Mobility Degradation at Interfaces*

AutoOrientation		– Use a parameter set based on the orientation of the nearest interface. <a href="#">Auto-Orientation for the Lombardi Model on page 407</a>
ParameterSetName	=<string>	Name of <code>EnormalDependence</code> parameter set. <a href="#">Named Parameter Sets for the Lombardi Model on page 407</a>

---

## Radiation Models

Table 282 AlphaParticle() *Carrier Generation by Alpha Particles*

Direction	=<vector>	! Direction of motion of the particle.
Energy	=<float>	! eV Energy of the alpha particle.
Location	=<vector>	! $\mu\text{m}$ Point where the alpha particle enters the device (bidirectional track).
StartPoint	=<vector>	! $\mu\text{m}$ Point where the alpha particle enters the device (one-directional track).
Time	=<float>	! s Time at which the charge generation peaks.

Table 283 Heavylon() *Carrier Generation by Heavy Ions*

Direction	=<vector>	Direction of motion of the ion.
Exponential		* Use exponential shape for the spatial distribution $R(w)$ .
Gaussian		Use Gaussian shape for the spatial distribution $R(w)$ .
Length	=[<float>...]	Length $l$ where <code>wt_hi</code> and <code>LET_f</code> are specified (in cm by default or $\mu\text{m}$ if the <code>PicoCoulomb</code> option is selected).
LET_f	=[<float>...]	Linear energy transfer (LET) function (in pairs $\text{cm}^{-3}$ by default or $\text{pC } \mu\text{m}^{-1}$ if the <code>PicoCoulomb</code> option is selected). Entries in the list match those in <code>Length</code> .
Location	=<vector>	$\mu\text{m}$ Point where the heavy ion enters the device (bidirectional track).
PicoCoulomb		Switch units for <code>LET_f</code> , <code>wt_hi</code> , and <code>Length</code> .

## Appendix G: Command File Overview

### Physics

Table 283 *Heavylon()* [Carrier Generation by Heavy Ions \(Continued\)](#)

SpatialShape	=<ident> [ ( <a href="#">Table 364</a> ) ]	PMI for spatial distribution function. <a href="#">Heavy Ion Spatial Distribution on page 1462</a>
StartPoint	=<vector>	μm Point where the heavy ion enters the device (one-directional track).
Time	=<float>	s Time at which the ion penetrates the device.
Wt_hi	=[ <float>... ]	Characteristic distance, $w_t(l)$ (in cm by default or μm if the <a href="#">PicoCoulomb</a> option is selected). Entries in the list match those in Length.

Table 284 *Radiation()* [Chapter 22 on page 772](#)

Dose	=<float>	rad Total dose over exposure time.
DoseRate	=<float>	rads <sup>-1</sup> Dose rate.
DoseTime	=(<float>*2)	! s Exposure time.
DoseTSigma	=<float>	! s Standard deviation of rise and fall of dose rate over time.

---

## Various

Table 285 *Aniso()* [Chapter 28 on page 883](#)

Avalanche		Use anisotropic avalanche generation. <a href="#">Anisotropic Avalanche Generation on page 899</a>
direction	[ (<System_Coord>) ]	Crystal or simulation system coordinate.
	=(<float>*3)	Anisotropic direction. <a href="#">Anisotropic Direction in 3D Cases on page 887</a>
	=xAxis	Equivalent to direction=(1 0 0).
	=yAxis	Equivalent to direction=(0 1 0).
	=zAxis	Equivalent to direction=(0 0 1).
eAvalanche		Use anisotropic electron avalanche generation. <a href="#">Anisotropic Avalanche Generation on page 899</a>

## Appendix G: Command File Overview

### Physics

Table 285 Aniso() [Chapter 28 on page 883 \(Continued\)](#)

eMobility		Use self-consistent anisotropic mobility for electrons. <a href="#">Self-Consistent Anisotropic Mobility on page 896</a>
eMobilityFactor	(Total)=<float>	Total anisotropic mobility factor for electrons. <a href="#">Total Anisotropic Mobility on page 896</a>
eQuantumPotential		Use electron anisotropic density-gradient equation with tensor grid approximation. <a href="#">Density Gradient Model on page 362</a>
hAvalanche		Use anisotropic hole avalanche generation. <a href="#">Anisotropic Avalanche Generation on page 899</a>
hMobility		Use self-consistent anisotropic mobility for holes. <a href="#">Self-Consistent Anisotropic Mobility on page 896</a>
hMobilityFactor	(Total)=<float>	Total anisotropic mobility factor for holes. <a href="#">Total Anisotropic Mobility on page 896</a>
hQuantumPotential		Use hole anisotropic density-gradient equation with tensor grid approximation. <a href="#">Using the Density Gradient Model on page 363</a>
MetalResistivity		Use self-consistent anisotropic metal resistivity. <a href="#">Self-Consistent Anisotropic Metal Resistivity on page 899</a>
MetalResistivity Factor	(Total)=<float>	Total anisotropic metal resistivity factor. <a href="#">Total Anisotropic Metal Resistivity Factor on page 899</a>
Mobility		Use self-consistent anisotropic mobility. <a href="#">Self-Consistent Anisotropic Mobility on page 896</a>
Poisson		Use anisotropic electrical permittivity. <a href="#">Anisotropic Electrical Permittivity on page 901</a>
Temperature		Use anisotropic thermal conductivity. <a href="#">Anisotropic Thermal Conductivity on page 902</a>

## Appendix G: Command File Overview

### Physics

**Table 286 BandEdge() in RandomizedVariation() Band Edge Variations**

<a href="#">Table 230</a>		Statistical properties of the correlation.
<a href="#">Table 327</a>		Spatial restriction of variation.
Amplitude_Chia	=<float>	eV Amplitude of affinity variation for exponential and Gaussian correlation.
Amplitude_Eg	=<float>	eV Amplitude of bandgap variation for exponential and Gaussian correlation.
Chi2Eg	=<float>	0 Correlation between affinity and band gap.
ChiComponent	=<0..1>	0 Named random field component for affinity randomization.
EgComponent	=<0..1>	1 Named random field component for bandgap randomization.
GrainChi	=(<float>...)	eV Grain affinity values.
GrainEg	=(<float>...)	eV Grain bandgap values.
GrainProbability	=(<float>...)	Probability for grain types.
Volume	=<string>	Named volume in which to activate the variation.

**Table 287 BarrierType() in SingletExciton() Singlet Exciton Equation**

Bandgap		* (i) Use bandgap difference as barrier.
CondBand		(i) Use conduction band-edge difference as barrier.
ValBand		(i) Use valence band-edge difference as barrier.

**Table 288 ComplexRefractiveIndex() Complex Refractive Index Model**

CarrierDep(	imag	Use carrier dependency of extinction coefficient. <a href="#">Carrier Dependency on page 695</a>
	real)	Use carrier dependency of refractive index. <a href="#">Carrier Dependency on page 695</a>

## Appendix G: Command File Overview

### Physics

**Table 288** *ComplexRefractiveIndex()* *Complex Refractive Index Model (Continued)*

CRIModel(	Name=<string>)	Activate user-defined complex refractive index model. <a href="#">Complex Refractive Index Model Interface on page 1360</a>
GainDep	(real(lin))	Use linear gain dependency of refractive index. <a href="#">Gain Dependency on page 697</a>
	(real(log))	Use logarithmic gain dependency of refractive index. <a href="#">Gain Dependency on page 697</a>
TemperatureDep(	real)	Use temperature dependency of refractive index. <a href="#">Temperature Dependency on page 695</a>
WavelengthDep(	imag	Use wavelength dependency of extinction coefficient. <a href="#">Wavelength Dependency on page 695</a>
	real)	Use wavelength dependency of refractive index. <a href="#">Wavelength Dependency on page 695</a>

**Table 289** *Conductivity() in RandomizedVariation()* *Metal Conductivity Variations*

<a href="#">Table 230</a>		Statistical properties of the correlation.
<a href="#">Table 327</a>		Spatial restriction of variation.
Amplitude	=<float>	A/cmV Amplitude of conductivity variation for exponential and Gaussian correlation.
GrainConductivity	=(<float>...)	A/cmV Grain conductivity values.
GrainProbability	=(<float>...)	Probability for grain types.
Volume	=<string>	Named volume in which to activate the variation.

**Table 290** *DeterministicVariation()* *Deterministic Variations*

DopingVariation	<string> ( <a href="#">Table 293</a> )	<a href="#">Deterministic Doping Variations on page 809</a>
GeometricVariation	<string> ( <a href="#">Table 307</a> )	<a href="#">Deterministic Geometric Variations on page 810</a>
ParameterVariation	<string> ( <a href="#">Table 322</a> )	<a href="#">Parameter Variations on page 811</a>

## Appendix G: Command File Overview

### Physics

**Table 291 Doping() in Noise() Random Dopant Fluctuations**

<a href="#">Table 327</a>		Spatial restriction of fluctuations.
BandgapNarrowing		Include effect on bandgap narrowing.
Mobility		Include effect on mobility.
Type	=Acceptor	Only acceptors fluctuate.
	=Donor	Only donors fluctuate.
	=Doping	* All dopants fluctuate.

**Table 292 Doping() in RandomizedVariation() Doping Variations**

<a href="#">Table 327</a>		Spatial restriction of variation.
BandgapNarrowing		+ Account for doping dependency of bandgap narrowing.
Mobility		+ Account for doping dependency of mobility.
Type	=Acceptor	Randomize acceptors only.
	=Donor	Randomize donors only.
	=Doping	* Randomize all dopants.

**Table 293 DopingVariation() in DeterministicVariation() Deterministic Doping Variations**

<a href="#">Table 327</a>		Spatial restriction of variation.
Amplitude	=<vector>	μm Vectorial amplitude for gradient specification.
Amplitude_Abs	=<vector>	μm Vectorial amplitude for absolute gradient specification.
Amplitude_Iso	=<float>	μm Isotropic amplitude for gradient specification.
BandgapNarrowing		+ Account for doping dependency of bandgap narrowing.
Conc	=<float>	cm <sup>-3</sup> Normalization concentration.
Factor	=<float>	1 Multiplier for variation.
Mobility		+ Account for doping dependency of mobility.

## Appendix G: Command File Overview

### Physics

Table 293 *DopingVariation() in DeterministicVariation()* [Deterministic Doping Variations](#)

SFactor	=<string>	Dataset name for variation.
Type	=Acceptor	Apply variation to acceptor concentration.
	=Donor	Apply variation to donor concentration.
	=Doping	* Apply variation according to sign to acceptor or donors.

Table 294 *DopingVariation() in RandomizedVariation()* [Doping Profile Variations](#)

<a href="#">Table 230</a>		Statistical properties of the correlation.
<a href="#">Table 327</a>		Spatial restriction of variation.
Amplitude	=<float>	0 μm Vectorial amplitude.
Amplitude_Iso	=<float>	0 μm Isotropic amplitude.
BandgapNarrowing		+ Account for doping dependency of bandgap narrowing.
Conc	=<float>	0 cm <sup>-3</sup> Normalization concentration.
Mobility		+ Account for doping dependency of mobility.
SFactor	=<string>	! Dataset name for variation.
Type	=Acceptor	Apply variation to acceptor concentration.
	=Donor	Apply variation to donor concentration.
	=Doping	* Apply variation according to sign to acceptor or donors.

## Appendix G: Command File Overview

### Physics

**Table 295    *EffectiveIntrinsicDensity()* Band Gap and Electron Affinity**

BandGap	(<ident> [( <a href="#">Table 364</a> )])	Use PMI model <ident> to compute band gap $E_g$ . <a href="#">Band Gap on page 1311</a>
BandGapNarrowing	(<ident> [( <a href="#">Table 364</a> )])	Use PMI model <ident> for bandgap narrowing. <a href="#">Bandgap Narrowing on page 1314</a>
	(BennettWilson)	* Bennett–Wilson model.
	(delAlamo)	del Alamo model.
	(JainRoulston)	Jain–Roulston model.
	(oldSlotboom)	Old Slotboom model.
	(Slotboom)	Slotboom model.
(TableBGN)		Table-based model.
NoBandGapNarrowing		No bandgap narrowing.
NoFermi		off Omit correction <a href="#">Equation 172</a> even when using Fermi statistics. <a href="#">Bandgap Narrowing With Incomplete Ionization on page 314</a>

**Table 296    *eLucky(), hLucky(), eFiegna(), hFiegna()* Effective Field**

CarrierTempDrive		Use field derived from temperature.
CarrierTempPost		Use field derived from postprocessed temperature.
Eparallel		* Use field parallel to interface.

**Table 297    *eNMP()* Extended Nonradiative Multiphonon Model**

Conc	=<float>	0 cm <sup>-2</sup> Precursor concentration for the eNMP model.
NumberOfSamples	=<int>	0 Number of random samples.

## Appendix G: Command File Overview

### Physics

**Table 297 eNMP() Extended Nonradiative Multiphonon Model (Continued)**

SFactor	=<ident> [ ( <a href="#">Table 364</a> ) ]	Name of the PMI model to compute the eNMP precursor concentration. <a href="#">Using the eNMP Model on page 628</a> and <a href="#">Space Factor on page 1479</a>
	=<string>	Dataset for the spatial distribution of the eNMP precursor concentration. <a href="#">Using the eNMP Model on page 628</a>

**Table 298 Epsilon() in RandomizedVariation() Dielectric Constant Variations**

<a href="#">Table 230</a>		Statistical properties of the correlation.
<a href="#">Table 327</a>		Spatial restriction of variation.
Amplitude	=<float>	1 Amplitude of relative dielectric constant variation for exponential and Gaussian correlation.
GrainEpsilon	=(<float>...)	1 Grain relative dielectric constant values.
GrainProbability	=(<float>...)	Probability for grain types.
Volume	=<string>	Named volume in which to activate the variation.

**Table 299 eQuantumPotential() and hQuantumPotential() Density Gradient Model**

AnisoAxes	[ (<System_Coord>) ]	Crystal or simulation system coordinate.
	={<vector> <vector>}	Two directions of anisotropy. <a href="#">Anisotropic Direction in 3D Cases on page 887</a>
AutoOrientation		off Use parameter set based on orientation of nearest interface. <a href="#">Auto-Orientation for the Density Gradient Model on page 367</a>
BoundaryCondition	=Dirichlet	(ci) Enforce homogeneous Dirichlet boundary condition.
	=HomogeneousNeumann	(ci) Enforce homogeneous Neumann boundary condition on region interfaces.
	=Neumann	(ci) Enforce homogeneous Neumann boundary condition.
Density		– (r) Use <a href="#">Equation 234</a> instead of <a href="#">Equation 235</a> .

## Appendix G: Command File Overview

### Physics

**Table 299 eQuantumPotential() and hQuantumPotential() Density Gradient Model**

direction	[ (<System_Coord>) ]	Crystal or simulation system coordinate.
	=(<float>*3)	Anisotropic direction. <a href="#">Anisotropic Direction in 3D Cases on page 887</a>
	=xAxis	Equivalent to direction=(1 0 0).
	=yAxis	Equivalent to direction=(0 1 0).
	=zAxis	Equivalent to direction=(0 0 1).
Formula	=<0..1>	0 (r) Select between formulation with mass <i>before</i> or <i>between</i> differential operators.
Gamma (	EffectiveMass	+ Quantization mass is density of states mass. <a href="#">Gamma Factor for Density Gradient Model on page 1457</a>
	name=<string> [ ( <a href="#">Table 364</a> ) ]	PMI for $\gamma$ . <a href="#">Gamma Factor for Density Gradient Model on page 1457</a>
Ignore		– (r) Compute, but do not apply quantum correction.
LocalModel	=<ident> [ ( <a href="#">Table 364</a> ) ]	(r) Name of PMI for apparent band-edge shift. <a href="#">Chapter 14 on page 348, Apparent Band-Edge Shift on page 1308</a>
	=SchenkBGN_elec	(r) Schenk bandgap narrowing model for electrons. <a href="#">Schenk Bandgap Narrowing Model on page 310</a>
	=SchenkBGN_hole	(r) Schenk bandgap narrowing model for holes. <a href="#">Schenk Bandgap Narrowing Model on page 310</a>
LocalModelOnly		– (r) Use with the LocalModel specification. Automatically sets $\gamma = 0$ in density gradient specification.
ParameterSetName	=<string>	Name of QuantumPotentialParameters parameter set. <a href="#">Named Parameter Sets for the Density Gradient Model on page 367</a>
Resolve		– (r) Use more accurate discretization of non-heterointerfaces.

## Appendix G: Command File Overview

### Physics

**Table 300      *eSHEDistributionl()*, *hSHEDistribution()* Spherical Harmonics Expansion Method**

AdjustImpurityScattering		+ Use an option to adjust impurity scattering to match the low-field mobility specified in the <code>Physics</code> section.
FullBand		- Use the full band structure with the default band-structure file.
	=<string>	! File name of the user-defined band-structure file.
RTA		- Use relaxation time approximation.

**Table 301      *ExternalBoltzmannSolver()* External Boltzmann Solver**

Carriers=(	Electron   Hole)	Use quantum correction for electrons or holes.
ContactName	=<string>	Name of contact for which Sentaurus Device sends the value of the current to the Subband-BTE solver of Sentaurus Device QTX for outputting current-voltage characteristics.
DampingLength	=<float>	0.005 μm Distance from volume enclosed by slices beyond which quantum correction and effective mobility, if activated, is deactivated.
MaxMismatch	=<float>	1e-4 μm Tolerance in matching of slice meshes and device mesh.
Mobility(	<carrier>	Activate extraction of effective mobility by Subband-BTE solver of Sentaurus Device QTX, which is applied to 3D domain enclosed by slices in Sentaurus Device. The specification of exactly one carrier type is mandatory.
	(-DisableInternalMobilityModels))	Enable Sentaurus Device mobility models so that the effective mobility extracted by the Subband-BTE solver of Sentaurus Device QTX can complement certain mobility models specified in the Sentaurus Device command file.
NumberOfSlices	=<2..>	* Number of slices.

## Appendix G: Command File Overview

### Physics

**Table 301** *ExternalBoltzmannSolver()* [External Boltzmann Solver \(Continued\)](#)

SBTECommandFile	=<string>	Start the Subband-BTE solver automatically using this command file.
Volume	=<string>	Domain in which to restrict quantum corrections and effective mobility, if activated.

**Table 302** *ExternalSchroedinger()* [External 2D Schrödinger Solver](#)

Carriers=(	Electron	Use quantum correction for electrons.
	Hole)	Use quantum correction for holes.
DampingLength	=<float>	0.005 μm Distance from volume enclosed by slices beyond which quantum correction is disabled.
MaxMismatch	=<float>	1e-4 μm Tolerance in matching of slice meshes and device mesh.
NumberOfSlices	=<2..>	* Number of slices.
SBandCommandFile	=<string>	Start Sentaurus Band Structure automatically, using this command file.
Volume	=<string>	Domain to which to restrict quantum corrections.

**Table 303** *GateCurrent()* [Chapter 24 on page 819](#), [Chapter 25 on page 844](#)

<ident>(	<carrier>... [ <a href="#">Table 364</a> ])	(i) Use PMI model <ident> to compute hot-carrier injection. <a href="#">Hot-Carrier Injection on page 1465</a>
DirectTunneling		(i) Schenk direct tunneling model. <a href="#">Direct Tunneling on page 822</a>
eFiegna	[ ( <a href="#">Table 296</a> ) ]	(i) Fiegna model for electrons. <a href="#">Fiegna Hot-Carrier Injection on page 850</a>
eLucky	[ ( <a href="#">Table 296</a> ) ]	(i) Lucky electron model. <a href="#">Classical Lucky Electron Injection on page 849</a>
eSHEDistribution		(i) SHE distribution model for electrons. <a href="#">SHE Distribution Hot-Carrier Injection on page 852</a>

## Appendix G: Command File Overview

### Physics

**Table 303** *GateCurrent()* [Chapter 24 on page 819](#), [Chapter 25 on page 844](#) (Continued)

Fowler		(i) Fowler–Nordheim tunneling. <a href="#">Fowler–Nordheim Tunneling on page 820</a>
	(EVB)	(i) Fowler–Nordheim valence band tunneling of electrons. <a href="#">Fowler–Nordheim Tunneling on page 820</a>
GateName	=<string>	(i) Electrode for monitoring gate current. <a href="#">Overview of Hot-Carrier Injection Models on page 844</a>
hFiegna	[ ( <a href="#">Table 296</a> ) ]	(i) Fiegna model for holes. <a href="#">Fiegna Hot-Carrier Injection on page 850</a>
hLucky	[ ( <a href="#">Table 296</a> ) ]	(i) Lucky hole model. <a href="#">Classical Lucky Electron Injection on page 849</a>
hSHEDistribution		(i) SHE distribution model for holes. <a href="#">SHE Distribution Hot-Carrier Injection on page 852</a>
InjectionRegion	=<string>	Regions to which hot carriers are injected. <a href="#">Destination of Injected Current on page 845</a>
Interface		(i) Activate carrier injection with explicitly evaluated boundary conditions for continuity equations during a transient ( <a href="#">Carrier Injection With Explicitly Evaluated Boundary Conditions for Continuity Equations on page 867</a> ) or if not in transient monitor interface current. <a href="#">Overview of Hot-Carrier Injection Models on page 844</a>

**Table 304** *Gaussian()* in *IntensityDistribution()* [Spatial Intensity Function Excitation](#)

Length	=(<float>,<float>)	Length of Gaussian decay, cannot specify together with Sigma.
PeakPosition	=(<float>,<float>)	Peak position of the Gaussian profile, in local coordinate of the defined shape function.
PeakWidth	=(<float>,<float>)	Width of the plateau of the modified Gaussian profile.
Scaling	=<float>	Scaling factor for the Gaussian profile.
Sigma	=(<float>,<float>)	Sigma of Gaussian decay, cannot specify together with Length.

## Appendix G: Command File Overview

### Physics

Table 305 *Geometric() in RandomizedVariation() Geometric Variations*

<a href="#">Table 230</a>		Statistical properties of the correlation.
<a href="#">Table 327</a>		Spatial restriction of variation.
Amplitude	=<vector>	0 μm Vectorial amplitude for displacement.
Amplitude_Iso	=<float>	0 μm Isotropic amplitude for displacement.
Options	=<0..1>	0 Select approximation specific to insulator position variations.
Surface	=<string>	! Name of varying surface.
WeightDielectric	=<float>	0 Interpolation coefficient between two approximations of dielectric term.
WeightQuantumPotential	=<float>	0.5 Interpolation coefficient between two approximations for density-gradient quantum-correction term.

Table 306 *GeometricFluctuations in Noise() Random Geometric Fluctuations*

<a href="#">Table 327</a>		Spatial restriction of variation.
Options	=<0..1>	0 Select approximation specific to insulator position variations.
WeightDielectric	=<float>	0 Interpolation coefficient between two approximations of dielectric term.
WeightQuantumPotential	=<float>	0.5 Interpolation coefficient between two approximations for density-gradient quantum-correction term.

Table 307 *GeometricVariation() in DeterministicVariation() Deterministic Geometric Variations*

<a href="#">Table 327</a>		Spatial restriction of variation.
Amplitude	=<vector>	μm Vectorial amplitude for displacement.
Amplitude_Iso	=<float>	μm Isotropic amplitude for displacement.

## Appendix G: Command File Overview

### Physics

**Table 307     GeometricVariation() in DeterministicVariation()** [Deterministic Geometric Variations \(Continued\)](#)

Options	=<0..1>	0 Select approximation specific to insulator position variations.
Surface	=<string>	! Name of varying surface.
WeightDielectric	=<float>	0 Interpolation coefficient between two approximations of dielectric term.
WeightQuantumPotential	=<float>	0.5 Interpolation coefficient between two approximations for density-gradient quantum-correction term.

**Table 308     HeatCapacity()** [Heat Capacity](#)

<ident>	[ ( <a href="#">Table 364</a> ) ]	PMI model <ident> for lattice heat capacity. <a href="#">Heat Capacity on page 1339</a>
Constant		Constant lattice heat capacity.
PMIModel(	<a href="#">Table 324</a> )	Use multistate configuration-dependent model.
TempDep		* Temperature-dependent lattice heat capacity.

**Table 309     HydrogenAtom(), HydrogenIon(), HydrogenMolecule(), HydrogenSpeciesA(), HydrogenSpeciesB(), HydrogenSpeciesC()** [Hydrogen Transport](#)

Alpha	=<ident>	Use PMI model for prefactor of the thermal diffusion term $\alpha_{\text{td}}$ .
Diffusivity	=<ident>	Use PMI model for $D_i \exp(-E_{di}/(kT))$ .

**Table 310     HydrogenDiffusion()** [Hydrogen Transport](#)

HydrogenAtom(	<a href="#">Table 309</a> )	Hydrogen atom specification.
HydrogenIon(	<a href="#">Table 309</a> )	Hydrogen ion specification.
HydrogenMolecule(	<a href="#">Table 309</a> )	Hydrogen molecule specification.
HydrogenReaction(	<a href="#">Table 311</a> )	<a href="#">Reactions Between Mobile Elements on page 614</a>

## Appendix G: Command File Overview

### Physics

Table 310 *HydrogenDiffusion()* *Hydrogen Transport* (Continued)

HydrogenSpeciesA(	<a href="#">Table 309</a> )	Hydrogen species A specification.
HydrogenSpeciesB(	<a href="#">Table 309</a> )	Hydrogen species B specification.
HydrogenSpeciesC(	<a href="#">Table 309</a> )	Hydrogen species C specification.

#### Note:

In Table 311,  $d$  is 3 for bulk reactions and 2 for interface reactions.

Table 311 *HydrogenReaction()* *Reactions Between Mobile Elements*

FieldFromMaterial	=<string>	(i) Material where the electric field is obtained.
FieldFromRegion	=<string>	(i) Region where the electric field is obtained.
ForwardReactionCoef	=<float>	0 /cm <sup>d</sup> s Forward reaction coefficient.
ForwardReactionEnergy	=<float>	0 eV Forward reaction activation energy.
ForwardReactionField Coef	=<float>	0 cm/V Forward reaction field coefficient.
LHSCoef		Define particle numbers to be removed.
(	Electron=<int>	0 Number of electrons to be removed.
	Hole=<int>	0 Number of holes to be removed.
	HydrogenAtom=<int>	0 Number of hydrogen atoms to be removed.
	HydrogenIon=<int>	0 Number of hydrogen ions to be removed.
	HydrogenMolecule=<int>	0 Number of hydrogen molecules to be removed.
	HydrogenSpeciesA=<int>	0 Number of hydrogen species A to be removed.
	HydrogenSpeciesB=<int>	0 Number of hydrogen species B to be removed.
	HydrogenSpeciesC=<int> )	0 Number of hydrogen species C to be removed.

## Appendix G: Command File Overview

### Physics

**Table 311     *HydrogenReaction()* *Reactions Between Mobile Elements (Continued)***

Material	=<string>	(i) Material where the recombination rate enters.
Region	=<string>	(i) Region where the recombination rate enters.
ReverseReactionCoef	=<float>	0 /cm <sup>d</sup> s Reverse reaction coefficient.
ReverseReactionEnergy	=<float>	0 eV Reverse reaction activation energy.
ReverseReactionField Coef	=<float>	0 cm/V Reverse reaction field coefficient.
RHSCoef		Define particle numbers to be created.
(	Electron=<int>	0 Number of electrons to be created.
	Hole=<int>	0 Number of holes to be created.
	HydrogenAtom=<int>	0 Number of hydrogen atoms to be created.
	HydrogenIon=<int>	0 Number of hydrogen ions to be created.
	HydrogenMolecule=<int>	0 Number of hydrogen molecules to be created.
	HydrogenSpeciesA=<int>	0 Number of hydrogen species A to be removed.
	HydrogenSpeciesB=<int>	0 Number of hydrogen species B to be removed.
	HydrogenSpeciesC=<int> )	0 Number of hydrogen species C to be removed.

**Table 312     *Incompletelonization()* *Chapter 13 on page 339***

Dopants	=<string>	Restrict incomplete ionization to dopants named in <string>. Dopant names are separated by spaces.
Model(	<ident>(<string> [Table 364]))	Use PMI model <ident> for species named in <string>. <a href="#">Incomplete Ionization on page 1472</a>

## Appendix G: Command File Overview

### Physics

Table 312 *Incompletelonization()* [Chapter 13 on page 339](#) (Continued)

Split(	Doping=<string>	Dopant that is redistributed into multiple lattice sites. <a href="#">Multiple Lattice Sites on page 345</a>
	Weights=(<float>...))	Occupation probabilities of the various lattice sites. <a href="#">Multiple Lattice Sites on page 345</a>

Table 313 *Mechanics()* [Mechanics Solver](#)

binary	=<string>	Name of the Sentaurus Interconnect binary. <a href="#">Mechanics Solver on page 1008</a>
command	=<string>	Sentaurus Interconnect Tcl commands. <a href="#">Mechanics Solver on page 1008</a>
initial_structure	=<string>	Initial structure for first call of Sentaurus Interconnect. <a href="#">Mechanics Solver on page 1008</a>
parameter	=<string>	Sentaurus Interconnect parameters. <a href="#">Mechanics Solver on page 1008</a>

Table 314 *MetalWorkfunction()* [Metal Workfunction](#)

Randomize(	AtInsulatorInterface	off (r) Randomize workfunction at metal–insulator vertices only. <a href="#">Metal Workfunction Randomization on page 299</a>
	AverageGrainSize=<float>	! (r) Average metal grain size. <a href="#">Metal Workfunction Randomization on page 299</a>
	GrainProbability=(<float>...)	! (r) Probabilities that grains will have a certain workfunction value. <a href="#">Metal Workfunction Randomization on page 299</a>
	GrainWorkfunction=(<float>...)	! eV (r) Workfunction values corresponding to the above probabilities. <a href="#">Metal Workfunction Randomization on page 299</a>
	RandomSeed=<int>	Seed for the random number generator. <a href="#">Metal Workfunction Randomization on page 299</a>
	UniformDistribution)	off (r) Attempt to evenly distribute uniformly sized grains. <a href="#">Metal Workfunction Randomization on page 299</a>

## Appendix G: Command File Overview

### Physics

*Table 314 MetalWorkfunction() Metal Workfunction (Continued)*

SFactor	=<string>	Dataset for the spatial distribution of metal workfunction. <a href="#">Metal Workfunction on page 298</a>
	=<ident> [( <a href="#">Table 364</a> )]	Name of model to be used by the PMI to compute the spatial distribution of metal workfunction. <a href="#">Space Factor on page 1479</a>
	[Factor=<float>]	! Scale factor for normalized SFactor values. <a href="#">Metal Workfunction on page 298</a>
	[Offset=<float>]	! Offset for raw or normalized SFactor values. <a href="#">Metal Workfunction on page 298</a>
Workfunction	=<float>	eV Metal workfunction. <a href="#">Metal Workfunction on page 298</a>
	=(<float> <float>*3) ...	eV, μm, μm, μm Metal workfunction–position quadruplets. <a href="#">Metal Workfunction on page 298</a>

*Table 315 Model(), options of stress-dependent models [Chapter 31 on page 935](#)*

Deformation Potential		Linear deformation potential model for computing band structure. <a href="#">Deformation of Band Structure on page 940</a>
(	ekp	k·p method for electron bands to account for shear strain components. <a href="#">Deformation of Band Structure on page 940</a>
	hkp	6 × 6 k·p method for hole bands. <a href="#">Deformation of Band Structure on page 940</a>
	Minimum	Compute conduction and valence band edges using minimum band energies. <a href="#">Deformation of Band Structure on page 940</a>
	multivalley)	Compute conduction and valance band edges using Multivalley band structure. <a href="#">Using Deformation Potential Model on page 943</a>

## Appendix G: Command File Overview

### Physics

Table 315 *Model(), options of stress-dependent models* [Chapter 31 on page 935](#) (Continued)

DOS(	eMass	Strained electron effective mass and DOS model. <a href="#">Strained Electron Effective Mass and DOS on page 945</a>
	hMass	Strained hole effective mass and DOS model. <a href="#">Strained Hole Effective Mass and DOS on page 948</a>
	hMass(AnalyticLTFit)	Strained hole effective mass and DOS model with analytic lattice temperature fit. <a href="#">Strained Hole Effective Mass and DOS on page 948</a>
	hMass(NumericalIntegration ))	Strained hole effective mass and DOS model with numeric integrations. <a href="#">Strained Hole Effective Mass and DOS on page 948</a>

## Appendix G: Command File Overview

### Physics

**Table 315** *Model(), options of stress-dependent models* [Chapter 31 on page 935 \(Continued\)](#)

Mobility(	eFactor[ ( <a href="#">Table 328</a> ) ]	Piezoresistive factor for electrons. <a href="#">Isotropic Factor Models on page 988</a>
	eMinorityFactor=<float>	1.0 Factor to scale stress effect for minority electrons. <a href="#">Stress Mobility Model for Minority Carriers on page 996</a>
	eMinorityFactor(Doping Threshold=<float>)=<float>	1.0 Factor to scale stress effect for minority electrons with no doping dependency. <a href="#">Stress Mobility Model for Minority Carriers on page 996</a>
	eSaturationFactor=<float>	1.0 Saturation factor for electrons. <a href="#">Dependency of Saturation Velocity on Stress on page 997</a>
	eSubband(BalMob)	Stress-induced change of the electron ballistic mobility. <a href="#">Multivalley Ballistic Mobility Model on page 968</a>
	eSubband(Doping)	Strain-induced subband model for electrons with doping dependency. <a href="#">Multivalley Electron Mobility Model on page 953</a>
	eSubband(EffectiveMass))	Stress-induced change of the electron effective mass. <a href="#">Effective Mass on page 956</a>
	eSubband(EffectiveMass(-Transport))	Exclude 2D inverse transport mass tensor transformation. <a href="#">Effective Mass on page 956</a>
	eSubband(EffectiveMass(Transport<vector>))	Activate 1D inverse transport mass tensor transformation. <a href="#">Effective Mass on page 956</a>
	eSubband(EqRefMob)	Activate the equilibrium reference mobility computation. <a href="#">Using Multivalley Transferred Carrier Mobility Model on page 971</a>
	eSubband(Fermi)	Strain-induced subband model for electrons with carrier concentration (Fermi statistics) dependency. <a href="#">Multivalley Electron Mobility Model on page 953</a>
	eSubband(NoStressInDDmob)	Account for only ballistic mobility. <a href="#">Using Multivalley Ballistic Mobility Model on page 969</a>
	eSubband(Scattering)	Strain-induced subband model for electrons with scattering. <a href="#">Intervalley Scattering on page 955</a>

## Appendix G: Command File Overview

### Physics

*Table 315 Model(), options of stress-dependent models* [Chapter 31 on page 935 \(Continued\)](#)

Mobility(	eSubband(Scattering(MLDA))	Strain-induced subband model with interface scattering. <a href="#">Intervalley Scattering on page 955</a>
	eSubband(-RelChDir110)	Use <100> channel direction in reference mobility. <a href="#">Using Multivalley Electron Mobility Model on page 960</a>
	eSubband(-AutoOrientation)	Use full tensor reference mobility. <a href="#">Using Multivalley Electron Mobility Model on page 960</a>
	eTensor[ ( <a href="#">Table 328</a> ) ]	Piezoresistive tensor for electrons. <a href="#">Piezoresistance Mobility Model on page 976</a>
	Factor[ ( <a href="#">Table 328</a> ) ]	Piezoresistive factor for electrons and holes. <a href="#">Isotropic Factor Models on page 988</a>
	hFactor[ ( <a href="#">Table 328</a> ) ]	Piezoresistive factor for holes. <a href="#">Isotropic Factor Models on page 988</a>
	hMinorityFactor=<float>	1.0 Factor to scale stress effect for minority holes. <a href="#">Stress Mobility Model for Minority Carriers on page 996</a>
	hMinorityFactor(Doping Threshold=<float>)=<float>	1.0 Factor to scale stress effect for minority holes with no doping dependency. <a href="#">Stress Mobility Model for Minority Carriers on page 996</a>
	hSaturationFactor=<float>	1.0 Saturation factor for holes. <a href="#">Dependency of Saturation Velocity on Stress on page 997</a>
	hSixband	Intel stress-induced model for holes. <a href="#">Intel Stress-Induced Hole Mobility Model on page 971</a>
	hSixband(Doping)	Intel stress-induced model for holes with doping dependency. <a href="#">Intel Stress-Induced Hole Mobility Model on page 971</a>
	hSixband(Fermi)	Intel stress-induced model for holes with carrier concentration (Fermi statistics) dependency. <a href="#">Intel Stress-Induced Hole Mobility Model on page 971</a>
	hSubband(BalMob)	Stress-induced change of the hole ballistic mobility. <a href="#">Multivalley Ballistic Mobility Model on page 968</a>

## Appendix G: Command File Overview

### Physics

*Table 315 Model(), options of stress-dependent models* [Chapter 31 on page 935 \(Continued\)](#)

Mobility(	hSubband(Doping)	Strain-induced subband model for holes with doping dependency. <a href="#">Multivalley Ballistic Mobility Model on page 968</a>
	hSubband(EffectiveMass)	Stress-induced change of the holes effective mass. <a href="#">Effective Mass on page 963</a>
	hSubband(EffectiveMass(Transport))	Activate 2D inverse transport mass tensor transformation. <a href="#">Effective Mass on page 963</a>
	hSubband(EffectiveMass(Transport<vector>))	Activate 1D inverse transport mass tensor transformation. <a href="#">Effective Mass on page 963</a>
	hSubband(EqRefMob)	Activate the equilibrium reference mobility computation. <a href="#">Using Multivalley Transferred Carrier Mobility Model on page 971</a>
	hSubband(Fermi)	Strain-induced subband model for holes with carrier concentration (Fermi statistics) dependency. <a href="#">Multivalley Hole Mobility Model on page 963</a>
	hSubband(NoStressInDDmob)	Account for only ballistic mobility. <a href="#">Using Multivalley Ballistic Mobility Model on page 969</a>
	hSubband(Scattering)	Strain-induced subband model for holes with bulk scattering. <a href="#">Scattering on page 964</a>
	hSubband(Scattering(MLDA))	Strain-induced subband model for holes with interface scattering. <a href="#">Scattering on page 964</a>
	hSubband(-RelChDir110)	Use <100> channel direction in reference mobility. <a href="#">Using Multivalley Hole Mobility Model on page 966</a>
	hSubband(-AutoOrientation)	Use full tensor reference mobility. <a href="#">Using Multivalley Hole Mobility Model on page 966</a>
	hTensor[ ( <a href="#">Table 328</a> ) ]	Piezoresistive tensor for holes. <a href="#">Piezoresistance Mobility Model on page 976</a>
	SaturationFactor=<float>	1.0 Saturation factor for electrons and holes. <a href="#">Dependency of Saturation Velocity on Stress on page 997</a>
	Tensor[ ( <a href="#">Table 328</a> ) ]	Piezoresistive tensor for electrons and holes. <a href="#">Piezoresistance Mobility Model on page 976</a>

## Appendix G: Command File Overview

### Physics

**Table 316 MoleFraction() Mole-Fraction Specification**

Grading		Alternative way to specify grading; allows a nonzero mole fraction and different distance of grading from different parts of the boundaries.
((	GrDistance=<float>	μm Distance in the direction normal to the specified interface, where linear interpolation of mole fractions from the constant value to the specified boundary mole fractions occurs.
	RegionInterface=(<string>*2)	Restrict this grading to the given interface (applied to all interfaces by default).
	xFraction=<[0,1]>	Boundary xMoleFraction.
	yFraction=<[0,1]>....)	Boundary yMoleFraction.
GrDistance	=<float>	μm Distance in the direction normal to the boundaries of the specified regions where linear interpolation of mole fractions from the specified constant value to 0 occurs.
RegionName	=[ <string>....]	List of regions where the mole-fraction specification will take effect.
	=<string>	Regions where the mole-fraction specification will take effect.
xFraction	=<[0,1]>	Constant value of xMoleFraction.
yFraction	=<[0,1]>	Constant value of yMoleFraction.

**Table 317 MSConfig() Chapter 18 on page 584**

BandEdgeShift(	<ident> [( <a href="#">Table 364</a> )] [<int>])	Compute band-edge shifts for conduction and valence by PMI model <ident> with optional constructor argument <int>. <a href="#">Apparent Band-Edge Shift on page 595</a>
Conc	=<float>	0 cm <sup>-d</sup> Concentration of states ( $d = 3$ for bulk; $d = 2$ for interface MSCs).
eBandEdgeShift(	<ident> [( <a href="#">Table 364</a> )] [<int>])	Compute band-edge shift for conduction band by PMI model <ident> with optional constructor argument <int>. <a href="#">Apparent Band-Edge Shift on page 595</a>

## Appendix G: Command File Overview

### Physics

*Table 317 MSConfig() Chapter 18 on page 584* (Continued)

Elimination		+ Switch solving algorithm.
hBandEdgeShift(	<ident> [( <a href="#">Table 364</a> )] [<int>])	Compute band-edge shift for valence band by PMI model <ident> with optional constructor argument <int>. <a href="#">Apparent Band-Edge Shift on page 595</a>
Name	=<string>	! Identifier of <code>MSConfig</code> .
State(		! Define a state (at least two required).
	Charge=<int>	o Number of positive elementary charges.
	Hydrogen=<int>	o Number of hydrogen atoms.
	Name=<string>	! Identifier of state.
	TTOM)	– Activate trap occupancy for the given state.
Transition(	<a href="#">Table 330</a> )	! Define a transition.

*Table 318 NBTI() Two-Stage NBTI Degradation Model*

Conc	=<float>	0 cm <sup>-2</sup> Concentration of NBTI traps.
NumberOfSamples	=<int>	o Number of random samples.
hSHEDistribution		– Use hole-energy distribution function.

*Table 319 Noise() Chapter 23 on page 781*

BandEdgeFluctuations	<string> ( <a href="#">Table 327</a> )	Band edge fluctuations. <a href="#">Random Band Edge Fluctuations on page 794</a>
Conductivity Fluctuations	<string> ( <a href="#">Table 327</a> )	Metal conductivity fluctuations. <a href="#">Random Band Edge Fluctuations on page 794</a>

## Appendix G: Command File Overview

### Physics

**Table 319** *Noise()* [Chapter 23 on page 781](#) (Continued)

DiffusionNoise(	<a href="#">Table 327</a>	Diffusion noise, spatially restricted. <a href="#">Diffusion Noise on page 787</a>
	e_h_Temperature	Noise temperatures are the respective carrier temperatures.
	eTemperature	Noise temperature is electron temperature for electrons, lattice temperature for holes.
	hTemperature	Noise temperature is lattice temperature for electrons, hole temperature for holes.
	LatticeTemperature)	* Noise temperature is lattice temperature.
Doping(	<a href="#">Table 291</a> )	Random dopant fluctuations
EpsilonFluctuations	<string> ( <a href="#">Table 327</a> )	Dielectric constant fluctuations. <a href="#">Random Dielectric Constant Fluctuations on page 796</a>
FlickerGRNoise(	<a href="#">Table 327</a> )	Bulk flicker noise. <a href="#">Bulk Flicker Noise on page 788</a>
GeometricFluctuations	<string> ( <a href="#">Table 306</a> )	Geometric fluctuations for surface <string>. <a href="#">Random Geometric Fluctuations on page 790</a>
MonopolarGRNoise(	<a href="#">Table 327</a> )	Equivalent monopolar noise. <a href="#">Equivalent Monopolar Generation–Recombination Noise on page 788</a>
TrapConcentration(	<a href="#">Table 327</a> )	Trap concentration fluctuations. <a href="#">Random Trap Concentration Fluctuations on page 793</a>
Traps(	<a href="#">Table 327</a> )	Trapping noise. <a href="#">Trapping Noise on page 789</a>
Workfunction Fluctuations	<string> ( <a href="#">Table 327</a> )	Workfunction fluctuations for surface <string>. <a href="#">Random Workfunction Fluctuations on page 793</a>

## Appendix G: Command File Overview

### Physics

**Table 320      Optics() Transfer Matrix Method and Beam Propagation Method**

<a href="#">Table 274</a>		Optics standalone.
BPMScalar(	<a href="#">Table 240</a> )	Scalar beam propagation method (BPM) solver. <a href="#">Beam Propagation Method on page 756</a>
TMM(	<a href="#">Table 268</a> )	Transfer matrix method (TMM) solver. <a href="#">Transfer Matrix Method on page 734</a>

**Table 321      Optics() Specifying the Type of Optical Generation Computation**

ComplexRefractiveIndex(	<a href="#">Table 288</a> )	Complex refractive index models. <a href="#">Complex Refractive Index Model on page 694</a>
Excitation(		Specification of excitation parameters. <a href="#">Setting the Excitation Parameters on page 674</a>
	fromBottom   fromTop	Growth orientation from the bottom or the top of the device structure. <a href="#">Common Illumination Configurations on page 680</a>
	Intensity=<float>	Wcm <sup>-2</sup>
	Phi=<float>	0 deg Angle between projection of propagation direction on xy plane and x-axis.
	Polarization=<float>	0 [0,1] Definition of polarization as in <a href="#">Using Transfer Matrix Method on page 739</a> .
	Polarization=<ident>	<sup>TM</sup> Transverse electric (TE) or transverse magnetic (TM) polarized light.
	PolarizationAngle=<float>	0 deg Angle between H-field and $\hat{z} \times \hat{k}$ used for vectorial solvers only.
	ReferenceSurface=IlluminationWindow   PropagationDirectionNormal )	PropagationDirectionNormal Reference surface with respect to which the intensity is defined (only applies to TMM and OptBeam solvers). <a href="#">Setting the Excitation Parameters on page 674</a>

## Appendix G: Command File Overview

### Physics

**Table 321     Optics() Specifying the Type of Optical Generation Computation (Continued)**

Excitation(	Theta=<float>	0 deg Angle between propagation direction and z-axis.
	Wavelength=<float>	μm
	Window( <a href="#">Table 332</a> )	
OpticalGeneration(	<a href="#">Table 253</a> )	Optical generation models. <a href="#">Specifying the Type of Optical Generation Computation on page 649</a>
OpticalSolver (		Specification of optical solver method. <a href="#">Specifying the Optical Solver on page 669</a>
	BPM( <a href="#">Table 240</a> )	
	Composite	<a href="#">Composite Method on page 674</a>
	FDTD( <a href="#">Table 245</a> )	
	FromFile( <a href="#">Table 246</a> )	
	OptBeam( <a href="#">Table 252</a> )	
	RayTracing( <a href="#">Table 259</a> )	<a href="#">Raytracer on page 702</a>
	TMM( <a href="#">Table 268</a> )	Note that Excitation section must not be defined in TMM section. <a href="#">Using Transfer Matrix Method on page 739</a>
Verbosity	=<int>	1 Verbosity level for optical solver output. A verbosity level of 0 suppresses all optical solver output.

**Table 322     ParameterVariation() in DeterministicVariation() Parameter Variations**

Factor	=<float>	1 Multiplier for original parameter value.
Material	=<string>	Material location of varied parameter.
MaterialInterface	=<string>	Material interface location of varied parameter.
Model	=<string>	! Model to which varied parameter belongs.

## Appendix G: Command File Overview

### Physics

**Table 322** *ParameterVariation() in DeterministicVariation() Parameter Variations (Continued)*

Parameter	=<string>	! Name of the varied parameter.
Region	=<string>	Region location of varied parameter.
RegionInterface	=<string>	Region interface location of varied parameter.
Summand	=<float>	0 Summand to original parameter value.
Value	=<float>	Modified parameter value.

**Table 323** *Piezo() Chapter 31 on page 935*

Model(	= <a href="#">Table 315</a> )	Stress-dependent models. <a href="#">Deformation of Band Structure on page 940</a> to <a href="#">Mobility Modeling on page 952</a>
OriKddX	=<vector>	(1 0 0) Miller indices of the stress system relative to the simulation system.
OriKddY	=<vector>	(0 1 0) Miller indices of the stress system relative to the simulation system.
Strain	=(<float>*6)	Components $xx$ , $yy$ , $zz$ , $yz$ , $xz$ , $xy$ of strain tensor.
	=Hooke	Use Hooke's law to compute strain tensor from stress tensor. <a href="#">Equation 987 on page 936</a>
	=LoadFromFile	Load strain from Piezo file specified in File section.
Stress	=(<float>*6)	Pa Components $xx$ , $yy$ , $zz$ , $yz$ , $xz$ , $xy$ of stress tensor.
	=<ident> [( <a href="#">Table 364</a> )]	Use PMI model <ident> to compute stress. <a href="#">Stress on page 1388</a>

## Appendix G: Command File Overview

### Physics

**Table 324** *PMIModel() Multistate Configuration–Dependent Bulk Mobility, High-Field Saturation With Two Driving Forces, Multistate Configuration–Dependent Thermal Conductivity, Multistate Configuration–Dependent Heat Capacity*

Index	=<int>	0 Number passed to and interpreted by PMI model.
MSConfig	=<string>	"" Name of multistate configuration on which PMI depends.
Name	=<string>	! Name of PMI model.
String	=<string>	"" String passed to and interpreted by PMI model.
Table 364		PMI parameters.

**Table 325** *RandomizedVariation() in Physics{} Statistical Impedance Field Method*

BandEdge	<string> ( <a href="#">Table 286</a> )	– Band Edge Variations on page 805
Conductivity	<string> ( <a href="#">Table 289</a> )	– Metal Conductivity Variations on page 806
Doping()	<a href="#">Table 292</a> )	– Doping Variations on page 801
DopingVariation	<string> ( <a href="#">Table 294</a> )	– Doping Profile Variations on page 807
Epsilon	<string> ( <a href="#">Table 298</a> )	– Dielectric Constant Variations on page 807
Geometric	<string> ( <a href="#">Table 305</a> )	– Geometric Variations on page 804
TrapConcentration()	<a href="#">Table 327</a> )	– Trap Concentration Variations on page 802
Workfunction	<string> ( <a href="#">Table 333</a> )	– Workfunction Variations on page 803

**Table 326** *Schroedinger() 1D Schrödinger Equation*

DensityTail	=Extrapolate	* Use estimate for lowest noncomputed subband energy as $E_{\max, v}$ . <a href="#">Equation 232</a>
	=MaxEnergy	Use highest computed subband energy as $E_{\max, v}$ . <a href="#">Equation 232</a>
eDensityCorrection		+ Activate electron quantization model in the quantum well. <a href="#">1D Schrödinger Equation on page 351</a>
Electron		– Solve Schrödinger equation for electrons.

## Appendix G: Command File Overview

### Physics

**Table 326** *Schroedinger()* **1D Schrödinger Equation (Continued)**

EnergyInterval	[ (<carrier>) ]=float>	0 eV Highest energy to which eigensolutions are computed, measured from the lowest interior potential point on the nonlocal line on which the Schrödinger equation is solved. When 0, only bound solutions are computed. <a href="#">Using the 1D Schrödinger Equation on page 352</a>
Error	[ (<carrier>) ]=<(0 ,)>	1e-5 eV Precision target for eigenenergies. <a href="#">Using the 1D Schrödinger Equation on page 352</a>
hDensityCorrection		+ Activate hole quantization model in the quantum well. <a href="#">1D Schrödinger Equation on page 351</a>
Hole		- Solve Schrödinger equation for holes.
MaxSolutions	[ (<carrier>) ]=<0 ..>	5 Maximum number of eigensolutions computed per ladder. <a href="#">Using the 1D Schrödinger Equation on page 352</a>
Polarization	= TE   TM   Mixed	TE Set the polarization used for the computation of the optical transition matrix element. <a href="#">Optical Transition Matrix Element for Wurtzite Crystals on page 1086</a>
PolarizationFactor	=<[0 ,1]>	1 Set the polarization factor used for the computation of the optical transition matrix element in mixed polarization simulations. <a href="#">Optical Transition Matrix Element for Wurtzite Crystals on page 1086</a>
Smooth	=<float>	0 cm Length (measured from end of nonlocal line) over which to blend from classical to quantum density.

**Table 327** *Spatial restriction* [Energetic and Spatial Distribution of Traps, Options Common to sIFM Variations](#)

SpaceMid	=<vector>	(0 0 0) μm Center of spatial distribution.
SpaceSig	=<vector>	(1e100 1e100 1e100) μm Width of spatial distribution.
SpatialShape	=Gaussian	Use Gaussian shape function.
	=Uniform	* Use Uniform shape function.

## Appendix G: Command File Overview

### Physics

**Table 328** *Tensor(), eTensor(), hTensor() Piezoresistance Mobility Model, Factor(), eFactor(), hFactor() Isotropic Factor Models*

<ident>	[ (Table 364) ]	Use PMI model <ident> to compute piezoresistive prefactors (first-order piezoresistance tensor model only). <a href="#">Piezoresistive Coefficients on page 1386</a>
		Use PMI model <ident> to compute a mobility enhancement stress factor (isotropic factor model only). <a href="#">Mobility Stress Factor on page 1377</a>
ApplyToMobility Components		– Apply the Factor model enhancement to individual mobility components. <a href="#">Factor Models Applied to Mobility Components on page 995</a>
AutoOrientation		off Use parameter set based on orientation of nearest interface. <a href="#">Auto-Orientation for Piezoresistance on page 981</a> and <a href="#">Isotropic Factor Model Options on page 994</a>
ChannelDirection	=<1..3>	1 Channel direction (Factor models only). <a href="#">Isotropic Factor Models on page 988</a>
EffectiveStressModel		Use the effective stress Factor model. <a href="#">Effective Stress Model on page 990</a>
	(AxisAligned Normals)	off Use axis-aligned normals for the effective stress calculation. <a href="#">Effective Stress on page 991</a>
Enormal		off Use piezoresistive prefactors (first-order piezoresistance tensor model only). <a href="#">Enormal- and MoleFraction-Dependent Piezo Coefficients on page 981</a>
FirstOrder		* Use the first-order piezoresistance model. <a href="#">Piezoresistance Mobility Model on page 976</a>
Kanda		off Include temperature and doping dependency. <a href="#">Doping and Temperature Dependency on page 978</a>
ParameterSetName	=<string>	Name of Piezoresistance or EffectiveStressModel parameter set. <a href="#">Named Parameter Sets for Piezoresistance on page 981</a> and <a href="#">Isotropic Factor Model Options on page 994</a>
SecondOrder		Use the second-order piezoresistance model. <a href="#">Piezoresistance Mobility Model on page 976</a>

## Appendix G: Command File Overview

### Physics

**Table 328** *Tensor(), eTensor(), hTensor() Piezoresistance Mobility Model, Factor(), eFactor(), hFactor() Isotropic Factor Models (Continued)*

SFactor	=<string>	Dataset for the spatial distribution of the mobility enhancement factor. <a href="#">SFactor Dataset or PMI Model on page 994</a>
	=<ident> [( <a href="#">Table 364</a> )]	Name of model to be used by the PMI to compute the spatial distribution of the mobility enhancement factor. <a href="#">SFactor Dataset or PMI Model on page 994</a> and <a href="#">Space Factor on page 1479</a>

**Table 329** *ThermalConductivity() Thermal Conductivity*

<ident>	[( <a href="#">Table 364</a> )]	Use PMI model <ident> for thermal conductivity. <a href="#">Thermal Conductivity on page 1350</a>
Constant	Conductivity	Use constant conductivity.
	Resistivity	Use constant resistivity.
Formula		Use the built-in strategy for thermal conductivity. <a href="#">Thermal Conductivity on page 1350</a>
PMIModel(	<a href="#">Table 324</a> )	Use multistate configuration-dependent model.
TempDep	Conductivity	Use <a href="#">Equation 1096 on page 1019</a> .
	Resistivity	Use <a href="#">Equation 1095 on page 1019</a> .

**Table 330** *Transition() in MSConfig() Chapter 18 on page 584*

CEModel(	<ident> [( <a href="#">Table 364</a> )] [<int>])	! Use PMI_TrapCaptureEmission model <ident> with optional constructor argument <int>.
FieldFromInsulator		– Use the insulator electric field instead of the semiconductor electric field for the MSCs defined at the semiconductor-insulator interface.
From	=<string>	! Interacting state.
Name	=<string>	! Identifier of transitions.

## Appendix G: Command File Overview

### Physics

*Table 330 Transition() in MSConfig() Chapter 18 on page 584 (Continued)*

Reservoirs(		List of reservoirs for particle conservation.
	<string> (Particles= <int>) ...)	Reservoir <string> and number of involved particles <int>.
To	=<string>	! Reference state.

#### Note:

In Table 331,  $d = 3$  for bulk traps and  $d = 2$  for interface traps.

*Table 331 Traps(...) Chapter 17 on page 543, Chapter 19 on page 600*

Table 327		Spatial restriction of trap distribution.
Acceptor		Acceptor trap type. <a href="#">Trap Types on page 544</a>
ActEnergy	=<float>	eV Equilibrium activation energy of hydrogen on Si-H bonds, $\varepsilon_A^0$ . <a href="#">Chapter 19 on page 600</a>
Add2TotalDoping(		Include the trap concentration in the acceptor, donor, and total doping concentrations used to compute mobility, lifetimes, and so on. <a href="#">Specifying Doping Species on page 60</a>
	ChargedTraps)	Include the charged trap concentration in the acceptor or donor doping concentrations used to compute mobility. <a href="#">Specifying Doping Species on page 60</a>
BarrierTunneling Enhan	=(<float>*4)	(1 1 1 1) 1, 1, 1, 1 Barrier tunneling enhancement parameters for the Degradation model depassivation constant, $\delta_{BTe}$ , $\rho_{BTe}$ , $\delta_{BTh}$ , and $\rho_{BTh}$ . <a href="#">Reaction Enhancement Factors on page 605</a>
BondConc	=<float>	$\text{cm}^{-d}$ Total silicon dangling bond concentration $N$ . <a href="#">Chapter 19 on page 600</a>
BondConcSFactor	=<ident> [(Table 364)]	Name of model to be used by the PMI to compute the spatial distribution of bond concentration. <a href="#">Space Factor on page 1479</a>
	=<string>	Dataset for the spatial distribution of bond concentration. <a href="#">Using the Trap Degradation Model on page 606</a>

## Appendix G: Command File Overview

### Physics

Table 331 *Traps(...)* [Chapter 17 on page 543](#), [Chapter 19 on page 600](#) (Continued)

CBRate	=<ident> [( <a href="#">Table 364</a> )]	Use PMI <ident> to compute electron capture and emission rate for conduction band.
	=(<ident> [( <a href="#">Table 364</a> )] <int>)	Use PMI <ident> with constructor argument <int> to compute electron capture and emission rate for conduction band. <a href="#">Trap Capture and Emission Rates on page 1392</a>
Conc	=<float>	cm <sup>-d</sup> or eV <sup>-1</sup> cm <sup>-d</sup> Concentration or the peak density of the trap distribution. For KMC-MIM simulation, sets defect density. <a href="#">Energetic and Spatial Distribution of Traps on page 544</a> , <a href="#">Specifying Defects on page 1108</a>
Coupled	=Off	* Disable trap coupling.
	=Tunneling	Couple traps by tunneling. <a href="#">Trap-to-Trap Tunneling (DiscreteTrapT2T) on page 561</a>
CritConc	=<float>	cm <sup>-d</sup> Critical concentration $N_{\text{crit}}$ , default 0.1N. <a href="#">Chapter 19 on page 600</a>
CurrentEnhan	=(<float>*4)	(0 1 0 1) 1, 1, 1, 1 Parameters of the tunneling and hot carrier-dependent terms of the depassivation constant, $\delta_{\text{Tun}}$ , $\rho_{\text{Tun}}$ , $\delta_{\text{HC}}$ , and $\rho_{\text{HC}}$ . <a href="#">Reaction Enhancement Factors on page 605</a>
Cutoff	=BandGap	Truncate trap energy distribution to plain band gap at 300 K. <a href="#">Energetic and Spatial Distribution of Traps on page 544</a>
	=EffectiveBandgap	* Truncate trap energy distribution to effective bandgap at 300 K.
	=None	Do not truncate trap energy distribution.
	=Simple	Use a legacy truncation of trap energy distribution.
Degradation		Use degradation model based on kinetic equation. <a href="#">Chapter 19 on page 600</a>
	(PowerLaw)	Use degradation model based on power law. <a href="#">Chapter 19 on page 600</a>

## Appendix G: Command File Overview

### Physics

Table 331 *Traps(...)* [Chapter 17 on page 543](#), [Chapter 19 on page 600](#) (Continued)

DePasCoef	=<float>	$\text{s}^{-1}$ Depassivation coefficient $v_0$ at the passivation conditions (the equilibrium at the passivation temperature $T_0$ ). <a href="#">Chapter 19 on page 600</a>
DiffusionEnhan	= (6*<float>)	$\text{cm cm}^2\text{s}^{-1} \text{eV cms}^{-1} \text{cm}^{-3}$ Parameters of hydrogen diffusion in oxide, $x_p$ , $D_0$ , $\epsilon_H$ , $k_p$ , $N_H^0$ , and $N_{\text{ox}}$ in <a href="#">Equation 592 on page 603</a> , <a href="#">Chapter 19 on page 600</a> .
DirectTunneling Enhan	=(<float>*4)	(1 1 1 1) 1, 1, 1, 1 Direct tunneling enhancement parameters for the Degradation model depassivation constant, $\delta_{\text{DTe}}$ , $\rho_{\text{DTe}}$ , $\delta_{\text{DTh}}$ , and $\rho_{\text{DTh}}$ . <a href="#">Reaction Enhancement Factors on page 605</a>
DiscreteTrapT2T		Specify trap-to-trap tunneling entry. <a href="#">Trap-to-Trap Tunneling (DiscreteTrapT2T) on page 561</a>
Donor		Donor trap type. <a href="#">Trap Types on page 544</a>
eBarrierTunneling(	NonLocal=<string>	Use nonlocal tunneling from the conduction band at reference surface of all connected unnamed, and all listed, connected named nonlocal meshes. <a href="#">Nonlocal Tunneling for Traps on page 576</a>
	TwoBand )	Use two-band dispersion. <a href="#">WKB Tunneling Probability on page 833</a>
eConstEmissionRate	=<float>	$\text{s}^{-1}$ Constant electron emission rate term to the conduction band $e_{\text{const}}^n$ . <a href="#">Local Capture and Emission Rates on page 555</a>
eGfactor	=<float>	Electron degeneracy factor $g_n$ . <a href="#">Trap Occupation Dynamics on page 553</a>

## Appendix G: Command File Overview

### Physics

Table 331 *Traps(...)* [Chapter 17 on page 543](#), [Chapter 19 on page 600](#) (Continued)

eHCSDegradation		Use the hot-carrier stress (HCS) degradation model for electrons. <a href="#">Hot-Carrier Stress Degradation Model on page 631</a>
(	BDEnergyIntervals=<int>	20 Number of energy intervals to use in the bond dispersion integral. <a href="#">Bond Dispersion on page 635</a>
	BondDispersion	+ Use bond dispersion. <a href="#">Bond Dispersion on page 635</a>
	SHE)	- Use the carrier distribution function from SHE. <a href="#">Spherical Harmonics Expansion Option on page 634</a>
eJfactor	=<float>	Electron J-model factor $g_n^J$ . <a href="#">Local Capture and Emission Rates on page 555</a>
ElectricField	[ (<carrier>) ]	Field-dependent model for cross sections. <a href="#">J-Model for Cross Sections on page 556</a>
EmptyCharge	=<int>	1 Set the charge state of an empty defect for KMC-MIM simulation. Applies only when <code>IncludeMIMChargeInPoisson</code> is specified. <a href="#">Specifying Defects on page 1108</a> , <a href="#">Including the MIM Charge in the Poisson Equation on page 1122</a>
EnergyMid	=<float>	eV Central energy $E_0$ of the trap distribution. <a href="#">Equation 521 on page 545</a>
EnergyShift	=<ident> [( <a href="#">Table 364</a> )]	Use PMI <ident> to compute trap energy shift.
	=(<ident> [( <a href="#">Table 364</a> )] <int>)	Use PMI <ident> with constructor argument <int> to compute trap energy shift. <a href="#">Energetic and Spatial Distribution of Traps on page 544</a> , <a href="#">Trap Energy Shift on page 1396</a>
EnergySig	=<(0,)>	eV Width $E_S$ of the trap distribution. <a href="#">Equation 521 on page 545</a>
eNeutral		Electron trap type. <a href="#">Trap Types on page 544</a>
eSHEDistribution	=(<float>*6)	$(0 \ 0 \ 0 \ 0 \ 1 \ 1) \ (\text{cm}^2 \text{A}^{-1})^{\rho_{\text{SHE}}}$ , eV, eV, eV, 1, 1 Parameters of the electron energy-dependent terms of the depassivation constant, $\delta_{\text{SHE}}$ , $\varepsilon_{\text{th}}$ , $\varepsilon_a$ , $\delta_{\perp}$ , $\rho_{\perp}$ , and $\rho_{\text{SHE}}$ . <a href="#">Trap Degradation Model on page 601</a>

## Appendix G: Command File Overview

### Physics

Table 331 *Traps(())* [Chapter 17 on page 543](#), [Chapter 19 on page 600](#) (Continued)

EtEmpty	=<float>	eV Set the empty trap level measured from the conduction band for KMC-MIM simulation. <a href="#">Specifying Defects on page 1108</a>
EtFill	=<float>	eV Set filled trap level measured from the conduction band for KMC-MIM simulation. <a href="#">Specifying Defects on page 1108</a>
Exponential		Exponential energetic distribution. <a href="#">Energetic and Spatial Distribution of Traps on page 544</a> , <a href="#">Equation 521 on page 545</a>
eXsection	=<[ 0 , )>	cm <sup>2</sup> Electron capture cross section $\sigma_n^0$ . <a href="#">Local Capture and Emission Rates on page 555</a>
FieldEnhan	=(<float>*4)	( 0 1 0 1 ) eV cm <sup><math>\rho_{\parallel}</math></sup> V <sup>-<math>\rho_{\parallel}</math></sup> , 1, eV cm <sup><math>\rho_{\perp}</math></sup> V <sup>-<math>\rho_{\perp}</math></sup> , 1 Parameters of the electric field-dependent terms of the Si-H bond energy and the activation energy, $\delta_{\parallel}$ , $\rho_{\parallel}$ , $\delta_{\perp}$ , and $\rho_{\perp}$ . <a href="#">Chapter 19 on page 600</a>
FillFrac	=<float>	Initial fraction of defects that are filled with electrons in KMC-MIM simulations. <a href="#">Specifying Defects on page 1108</a>
FixedCharge		Fixed charge. <a href="#">Trap Types on page 544</a> , <a href="#">Insulator Fixed Charges on page 582</a>
FluenceDependence	=(<float>*3)	( 0 0 0 ) 1, 1, 1 Activate fluence degradation model with the given parameters $\beta_F$ , $f_{cb}$ , and $f_{vb}$ . <a href="#">Fluence Model on page 611</a>
FowlerNordheimEnhan	=(<float>*4)	( 1 1 ) 1, 1 Fowler–Nordheim tunneling enhancement parameters for the Degradation model depassivation constant, $\delta_{FN}$ , and $\rho_{FN}$ . <a href="#">Reaction Enhancement Factors on page 605</a>
fromCondBand		Zero point for $E_0$ is conduction band. <a href="#">Equation 522 on page 546</a>
fromMidBandGap		Zero point for $E_0$ is intrinsic energy. <a href="#">Equation 522 on page 546</a>
fromValBand		Zero point for $E_0$ is valence band. <a href="#">Equation 522 on page 546</a>

## Appendix G: Command File Overview

### Physics

Table 331 *Traps(())* [Chapter 17 on page 543](#), [Chapter 19 on page 600](#) (Continued)

gamma		1 1 Prefactor for elastic tunneling. <a href="#">Electron Capture Rate for the Elastic Transition on page 577</a>
Gaussian		Gaussian energetic distribution. <a href="#">Energetic and Spatial Distribution of Traps on page 544</a> , <a href="#">Equation 521 on page 545</a>
GrainBoundary		Generate traps on the grain boundary. A prerequisite is that the grain field must be generated with <code>GrainFieldGenerator</code> specified in the <code>Math</code> section. <a href="#">Specifying Defects on page 1108</a>
(	Density=<float>	$\text{cm}^{-2}$ for 3D device, $\text{cm}^{-1}$ for 2D device Defect density on the grain boundary. The calculated number of defects is then randomized over the insulator volume inside the KMC simulation space.
	EtEmpty=<float>	eV Set the empty grain-boundary trap level measured from the conduction band.
	EtFill=<float>	eV Set the filled grain-boundary trap level measured from the conduction band.
	FillFrac=<float>	Set the initial fraction of defects that are filled with electrons.
	TrapSigma=<float>)	eV Randomizes the grain-boundary trap energy levels by using a Gaussian distribution with this standard deviation.
hBarrierTunneling(	NonLocal=<string>	Use nonlocal tunneling from the valence band at reference surface of all connected unnamed, and all listed, connected named nonlocal meshes. <a href="#">Nonlocal Tunneling for Traps on page 576</a>
	TwoBand )	Use two-band dispersion. <a href="#">WKB Tunneling Probability on page 833</a>
hConstEmissionRate	=<float>	$\text{s}^{-1}$ Constant hole emission rate to the valence band $e_{\text{const}}^p$ . <a href="#">Local Capture and Emission Rates on page 555</a>
hGfactor	=<float>	Hole degeneracy factor $g_p$ . <a href="#">Trap Occupation Dynamics on page 553</a>

## Appendix G: Command File Overview

### Physics

Table 331 *Traps(...)* [Chapter 17 on page 543](#), [Chapter 19 on page 600](#) (Continued)

hHCSDegradation		Use the HCS degradation model for holes. <a href="#">Hot-Carrier Stress Degradation Model on page 631</a>
(	BDEnergyIntervals=<int>	20 Number of energy intervals to use in the bond dispersion integral. <a href="#">Bond Dispersion on page 635</a>
	BondDispersion	+ Use bond dispersion. <a href="#">Bond Dispersion on page 635</a>
	SHE)	- Use the carrier distribution function from SHE. <a href="#">Spherical Harmonics Expansion Option on page 634</a>
hJfactor	=<float>	Hole J-model factor $g_p^J$ . <a href="#">Local Capture and Emission Rates on page 555</a>
hNeutral		Hole trap type. <a href="#">Trap Types on page 544</a>
hSHEDistribution	=(<float>*6)	$(0 \ 0 \ 0 \ 0 \ 1 \ 1) \ (\text{cm}^2 \text{A}^{-1})^{\rho_{\text{SHE}}}$ , eV, eV, eV, 1, 1 Parameters of the hole energy-dependent terms of the depassivation constant, $\delta_{\text{SHE}}$ , $\varepsilon_{\text{th}}$ , $\varepsilon_a$ , $\delta_{\perp}$ , $\rho_{\perp}$ , and $\rho_{\text{SHE}}$ . <a href="#">Trap Degradation Model on page 601</a>
HuangRhys	=<[0,)>	Huang–Rhys factor. <a href="#">Nonlocal Tunneling for Traps on page 576</a>
hXsection	=<[0,)>	$\text{cm}^2$ Hole capture cross section $\sigma_p^0$ . <a href="#">Local Capture and Emission Rates on page 555</a>
KmcMim1   KmcMim2		Specify KMC-MIM trap type. <a href="#">Specifying Defects on page 1108</a>
Level		Trap with single energy level. <a href="#">Energetic and Spatial Distribution of Traps on page 544</a> , <a href="#">Equation 521 on page 545</a>
Location	=(<vector>...)	$\mu\text{m}$ Locations of traps coupled by tunneling. For KMC-MIM simulations, locations of specifically placed defects. <a href="#">Trap-to-Trap Tunneling (DiscreteTrapT2T) on page 561</a> , <a href="#">Specifying Defects on page 1108</a>
	=(mesh)	Select all vertices in the region for trap-to-trap tunneling. <a href="#">Trap-to-Trap Tunneling (DiscreteTrapT2T) on page 561</a>

## Appendix G: Command File Overview

### Physics

Table 331 *Traps(())* Chapter 17 on page 543, Chapter 19 on page 600 (Continued)

Makram-Ebeid	[ (<carrier> <simpleCapt>) ]	Use Makram-Ebeid-Lannoo model. <a href="#">Makram-Ebeid-Lannoo Model on page 559</a>
Material	=<string>	(i) Material to which energy specification refers. <a href="#">Energetic and Spatial Distribution of Traps on page 544</a>
Name	=<string>	Identifier for trap.
-NormalizeSfactor		Switch off normalization of Sfactor for DiscreteTrapT2T. Default is on. <a href="#">Trap-to-Trap Tunneling (DiscreteTrapT2T) on page 561</a>
NumberOfRandomTraps	=<int>	Input number of randomly distributed traps for trap-to-trap tunneling. <a href="#">Trap-to-Trap Tunneling (DiscreteTrapT2T) on page 561</a>
OutputT2TPaths	(<string>)	Name of input file to output trap-to-trap tunneling locations and linkages. <a href="#">Trap-to-Trap Tunneling (DiscreteTrapT2T) on page 561</a>
PasCoef	=<float>	$s^{-1}$ Passivation coefficient $\gamma_0$ . By default, computed automatically to provide the equilibrium, $v_0 N_{hb}^0 / (N - N_{hb}^0)$ . <a href="#">Chapter 19 on page 600</a>
PasTemp	=<float>	300 K Passivation temperature $T_0$ . <a href="#">Chapter 19 on page 600</a>
PasVolume	=<float>	0 cm <sup>2</sup>   cm <sup>3</sup> Passivation volume $\Omega$ . <a href="#">Chapter 19 on page 600</a>
PhononEnergy	=<[ 0 , )>	eV Phonon energy for inelastic tunneling. <a href="#">Nonlocal Tunneling for Traps on page 576</a>
PooleFrenkel		Use Poole-Frenkel model for exchange with conduction or valence band, depending on trap type. <a href="#">Poole-Frenkel Model for Cross Sections on page 557</a>
	(Electron)	Use Poole-Frenkel model for exchange with conduction band, irrespective of trap type.
	(Hole)	Use Poole-Frenkel model for exchange with valence band, irrespective of trap type.

## Appendix G: Command File Overview

### Physics

Table 331 *Traps(())* [Chapter 17 on page 543](#), [Chapter 19 on page 600](#) (Continued)

PowerEnhan	=(<float>*3)	(0 0 0) 1, V <sup>-1</sup> cm, V <sup>-1</sup> cm Parameters in the chemical potential for kinetic equation degradation and the power for degradation by power law, $\beta_0$ , $\beta_{//}$ , and $\beta_{\perp}$ . <a href="#">Chapter 19 on page 600</a>
Randomize	[=<int>]	Randomize traps. <a href="#">Trap Randomization on page 549</a>
RandomizeEnergy	[=<int>]	Randomize energy at each vertex for the Uniform, Gaussian, or Table trap energy distributions. <a href="#">Energetic and Spatial Distribution of Traps on page 544</a>
Reference	=BandGap	Refer trap energies to band edges excluding bandgap narrowing. <a href="#">Energetic and Spatial Distribution of Traps on page 544</a>
	=EffectiveBandgap	* Refer trap energies to effective band edges.
ReferencePoint	=<vector>	Coordinate where to take the data for EnergyShift.
Region	=<string>	(i) Region to which energy specification refers. <a href="#">Energetic and Spatial Distribution of Traps on page 544</a>
SFactor	=<string>	Dataset for the spatial distribution of the traps. <a href="#">Energetic and Spatial Distribution of Traps on page 544</a>
	=<ident> [(Table 364)]	Name of model to be used by the PMI to compute the spatial trap distribution. <a href="#">Space Factor on page 1479</a>
SimpleCapt		– Restrict field enhancement of trapping to the emission rates. <a href="#">Local Capture and Emission Rates on page 555</a>
SingleTrap		* Use a single trap. <a href="#">Specifying Single Traps on page 548</a>
	(RTN)	Specify as RTN trap. <a href="#">Specifying Single Traps on page 548</a>
T2TconcScale	=<float>	1 Scaling factor for DiscreteTrapT2T concentration. <a href="#">Trap-to-Trap Tunneling (DiscreteTrapT2T) on page 561</a>

## Appendix G: Command File Overview

### Physics

Table 331 *Traps(())* [Chapter 17 on page 543](#), [Chapter 19 on page 600](#) (Continued)

T2TfixedConc	=<float>	0 Fix the same concentration at every trap-to-trap vertex in the region. <a href="#">Trap-to-Trap Tunneling (DiscreteTrapT2T) on page 561</a>
T2Trecombination		* Switch on T2T recombination. <a href="#">Trap-to-Trap Recombination on page 570</a>
Table	=(<float>*2...)	eV eV <sup>-1</sup> cm <sup>-d</sup> Pairs of energy and concentration of a tabular approximation to a trap distribution. <a href="#">Energetic and Spatial Distribution of Traps on page 544</a> , <a href="#">Equation 521 on page 545</a>
Transition	= InelasticPhonon   ModeOscillator	ModeOscillator Choose the tunneling transition for trap-to-trap tunneling. <a href="#">Trap-to-Trap Tunneling (DiscreteTrapT2T) on page 561</a>
TrapSigma	=<float>	eV Randomize trap levels using a Gaussian distribution with this standard deviation for KMC-MIM simulation. <a href="#">Specifying Defects on page 1108</a>
TrapVolume	=<[ 0 , )>	μm <sup>3</sup> Interaction volume for nonlocal tunneling. <a href="#">Nonlocal Tunneling for Traps on page 576</a>
Tunneling(	Hurkx[(<carrier>)] )	Use Hurkx trap-assisted tunneling model. <a href="#">Hurkx Model for Cross Sections on page 557</a>
Uniform		Uniform energetic distribution. <a href="#">Energetic and Spatial Distribution of Traps on page 544</a> , <a href="#">Equation 521 on page 545</a>
VBRate	=<ident> [( <a href="#">Table 364</a> )]	Use PMI <ident> to compute hole capture and emission rate for valence band.
	=(<ident> [( <a href="#">Table 364</a> )] <int>)	Use PMI <ident> with constructor argument <int> to compute hole capture and emission rate for valence band. <a href="#">Trap Capture and Emission Rates on page 1392</a>
Xsection	=<float>	For KMC-MIM simulations, use to calculate defect localization radius used in inelastic electrode-to-trap rate calculations. <a href="#">Specifying Defects on page 1108</a>
XsectionT2T	=<float>	For KMC-MIM simulations, use to calculate defect localization radius used in multiphonon trap-to-trap calculations. <a href="#">Specifying Defects on page 1108</a>

## Appendix G: Command File Overview

### Physics

*Table 332 Window Illumination Window*

Circle(	Radius=<float>)	
Intensity Distribution(	Gaussian( <a href="#">Table 304</a> )	Specify spatial intensity profile. <a href="#">Spatial Intensity Function Excitation on page 684</a>
Line(	Dx=<float>	
	X1=<float>	
	X2=<float>)	
Origin	=(<float>*3)	Global location of window origin.
OriginAnchor	=<ident>	Center Origin anchor of illumination window in terms of cardinal direction (North, South, East, West, NorthEast, SouthEast, NorthWest, SouthWest).
Polygon(	(<float>*2)*n	Specification of simple polygon using several vertices.
	((<float>*2)*n)*m)	Specification of complex polygon using several loops of vertices.
Rectangle(	Dx=<float>	
	Dy=<float>	
	Corner1=(<float>*2)	
	Corner2=(<float>*2))	
RotationAngles	=(<float>*3)	Angles specifying global orientation of window coordinate system.

## Appendix G: Command File Overview

### Physics

**Table 332** Window *Illumination Window* (Continued)

WeightedAPD integration (		– Switch on weighted absorbed photon density (APD) integration. <a href="#">Accurate Absorbed Photon Density for 1D Optical Solvers on page 691</a>
	Mode= Auto   Full	Auto Set integration mode of APD.  Options are: <b>Auto</b> : Mixed wireframe and 1D integration. <b>Full</b> : Wireframe integration.
	NumberOfCellsPerLayer=<int>	100 Set the number of wireframe cells per layer.
	NumberOfTransverse CellsPerDirection=<int>	200 Wireframe discretization parameter of the illumination window plane.
	PrintInfo()	Print detailed information about wireframe integration.
XDirection	=(<float>*3)	(1, 0, 0) Direction of x-axis.
YDirection	=(<float>*3)	(0, 1, 0) Direction of y-axis.

**Table 333** *Workfunction()* in *RandomizedVariation()* *Workfunction Variations*

<a href="#">Table 230</a>		Statistical properties of the correlation.
<a href="#">Table 327</a>		Spatial restriction of variation.
Amplitude	=<float>	eV Amplitude of workfunction variation for exponential and Gaussian correlation.
GrainProbability	=(<(0,)>....)	Probability for grain orientation.
GrainWorkfunction	=(<float>....)	eV Grain workfunctions.
Surface	=<string>	! Surface of variation. <a href="#">Workfunction Variations on page 803</a>

## Appendix G: Command File Overview

### Plot

---

## Plot

Table 334 *Plot{} Device Plots*

<a href="#">Table 195</a>		Scalar plot data. <a href="#">Device Plots on page 177</a>
<a href="#">Table 195/Element</a>		Scalar plot datasets defined on elements.
<a href="#">Table 195/RegionInterface</a>		Scalar plot data on region interfaces; supported for just a few datasets. <a href="#">Interface Plots on page 180</a>
<a href="#">Table 195/Tensor</a>		Tensorial plot data. <a href="#">Device Plots on page 177</a>
<a href="#">Table 196/Vector</a>		Vectorial plot data. <a href="#">Device Plots on page 177</a>
<a href="#">Table 197/SpecialVector</a>		Special vectorial plot data; supported for the SHE distribution data. <a href="#">SHE Distribution Hot-Carrier Injection on page 852</a>
<a href="#">Table 199</a>		MIM particle data. <a href="#">Particle Plots on page 1140</a>
<a href="#">Table 335</a>		Occupation rates of <code>MSConfig</code> states. <a href="#">Specifying Multistate Configurations on page 586</a>
<code>DatasetsFromGrid</code>		Copy datasets from TDR grid file. <a href="#">What to Plot on page 177</a>
	(<ident>...)	Datasets to be copied.
<code>TrappingRates</code>		Trap capture and emission rates. <a href="#">Visualizing Traps on page 578</a>
<code>TTOMFraction</code>		Fraction of occupied traps. <a href="#">Transient Trap Occupancy Model on page 641</a>

Table 335 *MSConfig in Plot{} Specifying Multistate Configurations*

<code>MSConfig</code>		All state occupations of all <code>MSConfig</code> .
	(<string>)	All state occupations of <code>MSConfig &lt;string&gt;</code> .
	(<string1> <string2>)	Occupation of state <code>&lt;string2&gt;</code> of <code>MSConfig &lt;string1&gt;</code> .

## Appendix G: Command File Overview

### RayTraceBC

## RayTraceBC

Table 336 *RayTraceBC{} Boundary Condition for Raytracing*

Fresnel		Fresnel boundary condition.
Name	=<string>	Name of the reflectivity contact or photon-recycling contact.
LayerStructure{	<float> <string> [ ; <float> <string> ... ]	Definition of multilayer structure used for TMM calculation. First column contains thickness of layer in $\mu\text{m}$ . Second column contains material name of layer.
MapOptGenToRegions(	<string> <string> ...)	Specify list of regions to map the lumped TMM BC optical generation to.
PMIModel	=<ident> [(Table 364)]	Name of the PMI model associated with this BC contact.
QuantumEfficiency	=<float>	Define the quantum efficiency of the TMM BC optical generation.
ReferenceMaterial	=<string>	Definition of LayerStructure orientation. The topmost layer in the LayerStructure specification is connected to the region with material ReferenceMaterial.
ReferenceRegion	=<string>	Definition of LayerStructure orientation. The topmost layer in the LayerStructure specification is connected to the region with name ReferenceRegion.
Reflectivity	=<[0,1]>	0 Reflectivity.
{Side	= "X" Periodic}	Define periodic BC at opposing x-surfaces. <a href="#">Periodic Boundary Condition on page 722</a>
	= "Y" Periodic}	Define periodic BC at opposing y-surfaces.
	= "Z" Periodic}	Define periodic BC at opposing z-surfaces.
Transmittivity	=<[0,1]>	0 Transmittivity.

## Appendix G: Command File Overview

### Solve

## Solve

Table 337    *Solve{}*

<a href="#">Table 350</a>		Solve selected standalone problem.
[<ident>.] <a href="#">Table 342</a>		Solve selected equation [for instance <ident>].
ACCoupled( {	<a href="#">Table 338</a> )	Perform small-signal and noise analysis. <a href="#">Small-Signal AC Analysis on page 149</a>
	[<ident>.] <a href="#">Table 342...}</a>	Equations to be solved consistently.
Continuation( {	<a href="#">Table 339</a> )	Continuation options.
	<a href="#">Table 337</a> )	Problem to be solved.
Coupled( {	<a href="#">Table 340</a> )	Solve coupled equations consistently by Newton iteration.
	[<ident>.] <a href="#">Table 342...}</a>	Equations to be solved.
CurrentPlot	as Load	Control current output. <a href="#">When to Write to the Current File on page 162</a>
HBCoupled( {	<a href="#">Table 344</a> )	Perform harmonic balance analysis. <a href="#">Harmonic Balance Analysis Output on page 158</a>
	[<ident>.] <a href="#">Table 342...}</a>	Equations to be solved.
Load( 	<a href="#">Table 345</a> ) [ {	Load and continue with solution stored by <code>Save</code> . <a href="#">Save and Load Statements on page 213</a>
	Circuit	Load circuit.
	<ident>...} ]	Devices to be loaded.
NewCurrentPrefix	=<string>	Use prefix <string> for current file. <a href="#">NewCurrentPrefix Statement on page 164</a>
Plot( 	<a href="#">Table 345</a> )	Plot current solution. <a href="#">When to Plot on page 178</a>
	[ {<ident>...} ]	Devices to be plotted.

## Appendix G: Command File Overview

### Solve

*Table 337    Solve{} (Continued)*

Plugin( {	<a href="#">Table 347</a> )	Solve equations consistently by Gummel iteration.
	<a href="#">Table 350</a>	Solve selected standalone problem.
	[<ident>.] <a href="#">Table 342...</a>	Solve selected equation (for example, <ident>).
	Coupled ( <a href="#">Table 340</a> ) { [<ident>.] <a href="#">Table 342...</a> }....}	Solve coupled equations.
Quasistationary( {	<a href="#">Table 348</a> )	Quasistationary simulation options.
	<a href="#">Table 337</a> }	Problem to be solved.
Save	as Load	Save current solution for retrieval with Load. <a href="#">Save and Load Statements on page 213</a>
Set()	<a href="#">Table 349</a> )	Set status according to <a href="#">Table 349</a> .
SpectralPlot( [ {<ident>...} ]	<a href="#">Table 345</a> )	Plot spectral simulation results of current solution. <a href="#">When to Plot on page 178</a> , <a href="#">Illumination Spectrum on page 651</a>
		Devices to be plotted.
System	(<string>)	Execute UNIX command <string> (ignore return status). <a href="#">System Command on page 225</a>
+System	(<string>)	Execute UNIX command <string>; use its return status to decide whether it was successful. <a href="#">System Command on page 225</a>
Transient( {	<a href="#">Table 352</a> )	Transient simulation options.
	<a href="#">Table 337</a> }	Problem to be solved.
Unset	(<ident>...)	Unset nodes. <a href="#">Mixed-Mode Electrical Boundary Conditions on page 117</a>
	(TrapFilling)	Use the standard trap equations. <a href="#">Explicit Trap Occupation on page 550</a>

## Appendix G: Command File Overview

Solve

*Table 338 ACCoupled() Small-Signal AC Analysis*

<a href="#">Table 340</a>		Coupled() parameters.
ACCompute(	<a href="#">Table 345</a> )	Restrict AC or noise analysis to selected points in a Quasistationary command.
ACEExtract	=<string>	Prefix for the name of the file to which the results of AC analysis are written (overrides the specification from the <code>File</code> section).
ACMethod	=Blocked	Use Blocked solver.
ACPlot	=<string>	Plot the responses of the solution variables to the AC signals. <string> is a prefix for the names of the files to which the responses are written.
ACSubMethod	(<ident>)= <a href="#">Table 223</a>	Super (1D and 2D), ILS (3D) Inner solver for device <ident>.
CircuitNoise		Compute noise from circuit elements. <a href="#">Noise From SPICE Circuit Elements on page 796</a>
Decade		Use logarithmic intervals between the frequencies.
EndFrequency	=<(0,)>	! Hz Upper frequency.
Exclude	(<ident>...)	Devices that will be removed from the circuit for AC analysis.
Extraction{	<a href="#">Table 20</a> }	List of frequency-dependent extraction curves. <a href="#">Extraction File on page 184</a>
Linear		Use linear intervals between the frequencies.
Node	(<ident>...)	Nodes for which to perform AC analysis.
NoisePlot	=<string>	Prefix for a file name. <a href="#">Chapter 23 on page 781</a>
NumberOfPoints	=<0...>	! Number of frequencies for which to perform the analysis.
ObservationNode	(<ident>...)	Nodes for which to perform noise analysis; subset of those in <code>Node</code> . <a href="#">Chapter 23 on page 781</a>
Optical		Perform optical AC analysis. <a href="#">Optical AC Analysis on page 153</a>

## Appendix G: Command File Overview

### Solve

*Table 338 ACCoupled() Small-Signal AC Analysis (Continued)*

StartFrequency	=<(0,)>	! Hz Lower frequency.
VoltageGreen Functions		– Compute voltage responses for frequency zero. <a href="#">Analysis at Frequency Zero on page 783</a>

*Table 339 Continuation() Continuation Command*

AcceptNewtonParameter		Apply relaxed Newton parameters. <a href="#">Relaxed Newton Method on page 198</a>
BreakCriteria{	<a href="#">Table 213}</a>	Break criteria. <a href="#">Break Criteria: Conditionally Stopping the Simulation on page 119</a>
Decrement	=<float>	1.5 (2.0 with NewArc) Divisor for step size on failure to solve.
Iadapt	=<float>	0.0 A Lower current limit for adaptive algorithm.
Increment	=<float>	2.0 (1.5 with NewArc) Multiplier for step size on successful solve.
InitialVstep	=<float>	! Initial voltage step.
IscaleMin	=<float>	1e-10 A Value added to scale factor for current when using the NewArc option.
MaxCurrent	=<float>	! Upper current limit.
MaxIfactor	=<float>	1e35 Maximum-allowed current relative to previous point as a multiplication factor.
MaxIstep	=<float>	1e35 Maximum-allowed current step.
MaxRload	=<float>	1e14 Ω (1e8 with NewArc) Maximum value for adaptive load resistor.
MaxStep	=<float>	1e35 Maximum step for the internal arc length variable.
MaxVoltage	=<float>	! Upper voltage limit.
MaxVstep	=<float>	1e35 Maximum-allowed voltage step.
MinCurrent	=<float>	! Lower current limit.
MinRload	=<float>	1 W Minimum value for adaptive load resistor.

## Appendix G: Command File Overview

### Solve

*Table 339 Continuation() Continuation Command (Continued)*

MinStep	=<float>	1e-35 Minimum step for the internal arc length variable.
MinVoltage	=<float>	! Lower voltage limit.
Name	=<ident>	Electrode to bias in continuation. It must be specified.
NewArc		- Use a new arc length calculation for continuation.
NewGdiff		- (+ with NewArc) Use a new differential conductance calculation.
Rfixed	=<float>	0.001 Fixed resistor value.
RloadAdjust		- (+ with NewArc) Adjust the adaptive load resistance at low values of slope.
Vadapt	=<float>	0.0 V Lower voltage limit for adaptive algorithm.

*Table 340 Coupled() Coupled Statement*

CheckRhsAfterUpdate		Check whether the RHS can be reduced further after the update error has converged. If CheckRhsAfterUpdate is specified in the Math section, -CheckRhsAfterUpdate can be used to disable it for this Coupled statement.
Digits	=<float>	Relative error target.
IncompleteNewton		Incomplete Newton. <a href="#">Incomplete Newton Algorithm on page 196</a>
(	RhsFactor=<float>	Maximum change in RHS to allow old Jacobian to be reused.
	UpdateFactor=<float>)	Maximum change in update to allow old Jacobian to be reused.
Iterations	=<0...>	Maximum number of iterations of the Newton algorithm.
LineSearchDamping	=<(0,1]>	Minimal coefficient for line search damping. 1 disables damping. <a href="#">Damped Newton Iterations on page 195</a>
Method	=Table 223	Linear solver.
	=Blocked	Block decomposition solver.

## Appendix G: Command File Overview

### Solve

**Table 340** *Coupled()* *Coupled Statement* (Continued)

NotDamped	=<0..>	Number of Newton iterations before Bank–Rose damping is applied. <a href="#">Damped Newton Iterations on page 195</a>
PMI_NewtonStep	(<ident>)	Use PMI model <ident> for Newton step computation. <a href="#">Preprocessing for Newton Iterations and Newton Step Control on page 1446</a>
RhsAndUpdate Convergence		Require both RHS and update error convergence.
RhsMin	=<float>	Upper bound for RHS norm convergence.
RhsMinFactor	=<float>	Factor for RhsMin affecting convergence. <a href="#">Convergence and Error Control on page 193</a>
SubMethod	[(<ident>)]= <a href="#">Table 223</a>	Super (1D and 2D), ILS (3D) Inner solver (for device <ident>).
UpdateIncrease	=<float>	Maximum-allowed increase of update error per Newton step.
UpdateMax	=<float>	Maximum-allowed update error in Newton algorithm.

**Table 341** *Cyclic()* *Large-Signal Cyclic Analysis*

Accuracy	=<float>	Tolerance $\epsilon_{\text{cyc}}$ .
CyclicExtrapolation Method	=<int>	0 Cyclic extrapolation method. <a href="#">Description of Method on page 145</a>

## Appendix G: Command File Overview

### Solve

*Table 341 Cyclic() Large-Signal Cyclic Analysis (Continued)*

Extrapolate(	Average	+ Use averaged factor $r_{av}^o$ for each object. Factor $r$ is defined separately for each vertex.
	CMaxVal=<float>	25 Maximum-allowed value of extrapolation factor ( $r_{max}$ ) for circuit node voltages.
	Factor=<float>	1 Coefficient $f$ for $r_{av}^o$ estimation.
	Forward	– Proceed as in a standard transient, without cyclic extrapolation.
	MaxVal=<float>	25 Value of $r_{max}$ .
	MinVal=<float>	1 Value of $r_{min}$ .
	PMaxVal=<float>	25 Maximum-allowed value of extrapolation factor ( $r_{max}$ ) for electrostatic potential.
	Print	+ Print averaged factors $r_{av}^o$ for each object.
	QFtraps	– Apply extrapolation to ‘trap quasi-Fermi level’ $\Phi_T$ instead of trap occupation probabilities $f_T$ .
	TMaxVal=<float>)	25 Maximum-allowed value of extrapolation factor ( $r_{max}$ ) for carrier or lattice temperature.
IgnoreCircuit		– Ignore circuit node voltages from extrapolation.
IgnoreContact		– Ignore contact nodes from extrapolation.
IgnoreTemperaturePDE		– Exclude carrier temperature from extrapolation.
IgnoreDGPDE		– Exclude quantum potential from extrapolation.
IgnoreElectronPDE		– Exclude electron concentration from extrapolation.
IgnoreHolePDE		– Exclude hole concentration from extrapolation.
IgnoreLTemperaturePDE		– Exclude lattice temperature from extrapolation.

## Appendix G: Command File Overview

### Solve

**Table 341** *Cyclic()* [Large-Signal Cyclic Analysis](#) (Continued)

IgnorePotentialPDE		– Exclude electrostatic potential from extrapolation.
IgnoreRegion		– Ignore a particular region for extrapolation.
MaximumCycles	=<int>	100000 Number of cyclic cycles after which transient simulation methodology is switched on.
Period	=<float>	s Period of the cycle.
RegionName	=<string>	Name of region to be excluded from extrapolation.
RelFactor	=<float>	Relaxation factor $\gamma$ .
StartingPeriod	=<2..>	2 Period from which the extrapolation procedure starts.

In the description column of [Table 342](#), the default for `ErrRef` (first number; see [Table 211 on page 1573](#)) and its unit, and the default for `Error` (second number, see [Table 211](#)) are given.

**Table 342** *Equations*

Charge	1.602192e-19 C 1e-3 Floating gate charge, available for <code>TransientErrRef</code> and <code>TransientError</code> only. <a href="#">Floating Gates on page 1202</a>
Circuit	0.0258 V 1e-3 Circuit equations.
CondInsulator	0.0258 V 1e-3 Conductive insulator equations.
Contact	0.0258 V 1e-3 Contact equations.
Electron	1e10 cm <sup>-3</sup> 1e-5 Electron continuity equation. <a href="#">Chapter 8 on page 238</a>
eQuantumPotential	0.0258 V 1e-3 Electron quantum-potential equation. <a href="#">Density Gradient Model on page 362</a>
eSHEDistribution	1 V 1e-3 Electron SHE distribution equation. <a href="#">Using Spherical Harmonics Expansion Method on page 857</a>
eTemperature	300 K 1e-4 Electron temperature equation. <a href="#">Hydrodynamic Model for Temperatures on page 252</a>
FEPolarization	1e-9 C/cm <sup>2</sup> 1e-5 Ferroelectric polarization equation. <a href="#">Chapter 29 on page 905</a>
Hole	1e10 cm <sup>-3</sup> 1e-5 Hole continuity equation. <a href="#">Chapter 8 on page 238</a>

## Appendix G: Command File Overview

Solve

Table 342 Equations (Continued)

hQuantumPotential	0.0258 V 1e-3 Hole quantum potential equation. <a href="#">Density Gradient Model on page 362</a>
hSHEDistribution	1 V 1e-3 Hole SHE distribution equation. <a href="#">Using Spherical Harmonics Expansion Method on page 857</a>
hTemperature	300 K 1e-4 Hole temperature equation. <a href="#">Hydrodynamic Model for Temperatures on page 252</a>
HydrogenAtom	1e10 cm <sup>-3</sup> 1e-3 Hydrogen atom transport equation. <a href="#">Hydrogen Transport on page 612</a>
HydrogenIon	1e10 cm <sup>-3</sup> 1e-3 Hydrogen ion transport equation. <a href="#">Hydrogen Transport on page 612</a>
HydrogenMolecule	1e10 cm <sup>-3</sup> 1e-3 Hydrogen molecule transport equation. <a href="#">Hydrogen Transport on page 612</a>
HydrogenSpeciesA	1e10 cm <sup>-3</sup> 1e-3 Hydrogen species A transport equation. <a href="#">Hydrogen Transport on page 612</a>
HydrogenSpeciesB	1e10 cm <sup>-3</sup> 1e-3 Hydrogen species B transport equation. <a href="#">Hydrogen Transport on page 612</a>
HydrogenSpeciesC	1e10 cm <sup>-3</sup> 1e-3 Hydrogen species C transport equation. <a href="#">Hydrogen Transport on page 612</a>
LLG	Landau–Lifshitz–Gilbert equation. <a href="#">Chapter 30 on page 921</a>
Mechanics	Mechanical stress equation. <a href="#">Mechanics Solver on page 1008</a>
Poisson	0.0258 V 1e-3 Poisson equation. <a href="#">Electrostatic Potential on page 229</a>
SingletExciton	Singlet exciton equation. <a href="#">Singlet Exciton Equation on page 291</a>
TCircuit	Thermal circuit equations.
TContact	Thermal contact equations.
Temperature	300 K 1e-3 Temperature equation. <a href="#">Chapter 9 on page 245</a>
Traps	1e-5 1 1e-3 Explicit trap equation. <a href="#">Trap-to-Trap Tunneling (DiscreteTrapT2T) on page 561</a>

## Appendix G: Command File Overview

### Solve

**Table 343     Goal{} Quasistationary Ramps**

Charge	=<float>	C Target charge for contact.
Contact	=<string1>.<string2>	Name of instance and contact to be ramped.
Current	=<float>	A $\mu\text{m}^{d-3}$ Target current for contact.
Device	=<string>	Name of the device where the parameter will be ramped.
DopingWell	(<vector>)	Semiconductor well defined by point <vector> where quasi-Fermi potential will be ramped.
DopingWells	(Material=<string>)	Semiconductor wells in material <string> where quasi-Fermi potential will be ramped.
	(Region=<string>)	Semiconductor wells in the region <string> where quasi-Fermi potential will be ramped.
	(Semiconductor)	All device semiconductor wells where Fermi potential will be ramped.
eQuasiFermi	=<float>	V Target electron quasi-Fermi potential for semiconductor wells. <a href="#">Ramping Quasi-Fermi Potentials in Doping Wells on page 124</a>
hQuasiFermi	=<float>	V Target hole quasi-Fermi potential for semiconductor wells. <a href="#">Ramping Quasi-Fermi Potentials in Doping Wells on page 124</a>
Material	=<string>	Name of material where the parameter will be ramped.
MaterialInterface	=<string>	Name of material interface where parameter will be ramped.
Model	=<string>	Name of model for which the parameter will be ramped.
ModelParameter	=<string>	Parameter path or name of parameter that is ramped when using <a href="#">Parameter Ramping on page 689</a> .
Name	=<string>	Name of contact to be ramped.
	=Regexp(<string>)	Regular expression for matching contacts.
Node	=<string>	Name of node for which the voltage will be ramped.

## Appendix G: Command File Overview

### Solve

**Table 343     Goal{} Quasistationary Ramps (Continued)**

Parameter	=<string>	Name of parameter that will be ramped.
	=<string1>.<string2>	Name of instance and parameter that will be ramped. This syntax only applies to circuit parameters, but not device parameters.
Power	=<float>	Target heat for contact.
Region	=<string>	Name of region where the parameter will be ramped.
RegionInterface	=<string>	Name of region interface where parameter will be ramped.
Temperature	=<float>	K Target temperature for contact.
Value	=<float>	Target value for parameter.
Voltage	=<float>	V Target voltage for node or contact.
WellContactName	=<string>	Name of contact defining the well where quasi-Fermi potential will be ramped.

**Table 344     HBCoupled() Harmonic Balance**

Table 340		Coupled() parameters.
CNormPrint		+ Print instance equation errors per Newton step (MDFT and SDFT modes).
Derivative		- Use complete Jacobian for HB Newton.
GMRES		Use GMRES linear solver; requires Method=ILS.
(	MaxIterations =<int>	200 Maximal number of iterations.
	Restart=<int>	20 Size of minimization subspace.
	Tolerance=<float>)	1e-4 Required residuum reduction.

## Appendix G: Command File Overview

### Solve

*Table 344 HBCoupled() Harmonic Balance (Continued)*

Initialize	=DCMode	0-th harmonic from DC solution; others, zero.
	=HBMode	All harmonics from previous harmonic balance (HB) solution.
	=MixedMode	0-th harmonic from DC; others, from previous HB solution.
Method	=ILS	Required for GMRES.
	=Pardiso	Use PARDISO to solve linear system (SDFT mode only).
Name	=<string>	Name component of plot files. Default is Coupled_<number>, where <number> is the global index of all Coupled, ACCoupled, and HBCoupled solve entries.
RhsScale	( <a href="#">Table 342</a> )=<float>	Scaling of RHS in Newton (MDFT and SDFT modes).
Solve Spectrum	=<string>	Referenced solve spectrum (MDFT mode only).
Tone		Multiple specification for multitone (MDFT mode only).
(	Frequency=<freqspec>	! Hz Base frequency. <a href="#">Performing Harmonic Balance Analysis on page 155</a>
	NumberOfHarmonics=<int>)	! Number of harmonics $H$ . <a href="#">Equation 25 on page 154</a>
UpdateScale	( <a href="#">Table 342</a> )=<float>	Scaling of update in Newton (MDFT and SDFT modes).
ValueMin	( <a href="#">Table 342</a> )=<float>	Lower bound for quantity in time domain (MDFT mode only).
Value Variation	( <a href="#">Table 342</a> )=<float>	Allowed variation of quantity in time domain (MDFT mode only).

## Appendix G: Command File Overview

### Solve

**Table 345    Load(), Plot(), Save(), SpectralPlot() in Solve{} When to Plot**

Current	(Difference=<float>)	A Only for Continuation simulations. Write save or plot files when the difference between the actual and previous plotting current values on the continuation contact is greater than the specified Difference.
	(Intervals=<int>)	A Only for Continuation simulations. Divide the range specified with MinCurrent and MaxCurrent of Continuation into <int> intervals, and write save or plot files every time the current enters one of these intervals. When LogCurrent is specified in Continuation section, logarithmic (asinh( $10^{100}$ current)) current range is used. Write save or plot file when the current enters a new logarithmic interval.
	(LogDifference=<(0,)>)	A Only for Continuation simulations. LogCurrent must be specified in Continuation section. Write save or plot files when the difference between the actual and previous logarithmic plotting current values is greater than the specified Difference.
Explicit		– Write datasets specified in Plot section only. Explicit is ignored by SpectralPlot.
FilePrefix	=<string>	Prefix of file name. File names consist of the prefix, the instance name, an optional local number (depending on Overwrite and noOverwrite), and an extension. The default file prefix is save<globalsaveindex> for Save and plot<globalplotindex> for Plot.
Iterations	=(<int>;...)	(0;1;...) Save or plot at certain Plugin iterations.
IterationStep	=<int>	Save or plot all <int> steps of plugins, quasistationaries, continuations, or transients.
Loadable		+ Write additional information required to load a simulation from a .tdr file. Loadable is not supported for SpectralPlot.
noOverwrite		off Give each new save or plot file a new name by numbering.
Number	=<int>	Number of the solution to be retrieved by Load.
Overwrite		on Rewrite the same file name at each loop.

## Appendix G: Command File Overview

### Solve

**Table 345    Load(), Plot(), Save(), SpectralPlot() in Solve{} When to Plot (Continued)**

Time=(	<float>	Time point for plot. <a href="#">When to Plot on page 178</a>
	decade	Use logarithmic range subdivision. <a href="#">When to Plot on page 178</a>
	intervals=<int>	Number of subdivisions. <a href="#">When to Plot on page 178</a>
	range=(<float>*2); ...)	Plotting range. <a href="#">When to Plot on page 178</a>
Voltage	(Difference=<float>)	V Only for Continuation simulations. Write save or plot files when the difference between current and previous plotting voltages is greater than the specified Difference.
	(Intervals=<int>)	Only for Continuation simulations. Divide the range specified with MinVoltage and MaxVoltage of Continuation into <int> intervals, and write save or plot files every time the voltage enters one of these intervals.
(	When	Generate output whenever the target was crossed the between the previous and the current iteration $i$ , $X_{i-1} < X_T \leq X_i$ or $X_{i-1} > X_T \geq X_i$ . Available for Plot, SpectralPlot, Save, and CurrentPlot inside a Quasistationary, Transient, or Continuation.
	Contact=[<string>.]<string>	[Instance and] contact name for target. The instance name defaults to " ", appropriate for single-device simulation.
	Current=<float>	A Target current $X_T$ .
	Node=<string>	Node name for target.
	Voltage=<float>)	V Target voltage $X_T$ .

## Appendix G: Command File Overview

### Solve

**Table 346** *MSConfigs() in Set() in Solve{} Manipulating MSCs During Solve*

Frozen		+ Freeze MSCs.
MSConfig		Set occupations for the specified MSC.
(	Device=<string>	Device instance name to where the MSC belongs.
	Name=<string>	Name of MSC.
	State(Name=<string> Value=<float>)....)	Set occupation of MSC state <string> to given value.

**Table 347** *Plugin() Plugin Command*

BreakOnFailure		Stop when an inner Coupled fails.
Digits	=<float>	Relative precision target.
Iterations	=<int>	Maximum number of iterations. 0 is used to perform one loop without error testing.

**Table 348** *Quasistationary() Quasistationary Ramps*

AcceptNewtonParameter		Apply relaxed Newton parameters. <a href="#">Relaxed Newton Method on page 198</a>
(	ReferenceStep=<float>)	1.e-9 Apply relaxed Newton parameters for step size < ReferenceStep.
BreakCriteria{	<a href="#">Table 213</a> }	Break criteria. <a href="#">Break Criteria: Conditionally Stopping the Simulation on page 119</a>
Decrement	=<float>	2 Divisor for the step size when last step failed.
DoZero		+  – The equations are solved for $t = 0$ .
Extraction{	<a href="#">Table 20</a> }	List of voltage-dependent extraction curves. <a href="#">Extraction File on page 184</a>
Extrapolate	( <a href="#">Table 218</a> )	– Use extrapolation. <a href="#">Extrapolation on page 130</a>
Goal{	<a href="#">Table 343</a> }	Goal for ramping.
Increment	=<float>	2 Multiplier for the step size when last step was successful.

## Appendix G: Command File Overview

### Solve

**Table 348 Quasistationary() Quasistationary Ramps (Continued)**

InitialStep	=<float>	0.1 Initial step size.
MaxStep	=<float>	1 Maximum step size.
MinStep	=<float>	0.001 Minimum step size.
NewtonPlotStep	=<float>	Upper limit for step size for which to write Newton plot files. <a href="#">NewtonPlot on page 209</a>
NonlocalPath	( <a href="#">Table 227</a> )	<a href="#">Dynamic Nonlocal Path Band-to-Band Tunneling Model on page 529</a> , <a href="#">Handling Derivatives on page 535</a>
Plot{	Intervals=<int>	! Number of intervals in Range. <a href="#">Saving and Plotting During a Quasistationary on page 130</a>
	Range=(<float>*2)}	! Range of $t$ for which to generate plots.
PlotBandstructure	as Plot	Plot band structure data in an LED simulation.
PlotGain	as Plot	Plot stimulated and spontaneous emission data in LED simulation.
PlotLEDRadiation	as Plot	Optical far field of LED.
ReadExtrapolation		+ Try to use the extrapolation information from a previous Quasistationary if it is available and compatible. <a href="#">Extrapolation on page 130</a>
SaveOptField	as Plot	Save optical field data in an LED simulation.
SpectralPlot	as Plot	Plot spectral simulation results. <a href="#">Illumination Spectrum on page 651</a>
StoreExtrapolation		+ Store the extrapolation information internally at the end of the Quasistationary. <a href="#">Extrapolation on page 130</a>

## Appendix G: Command File Overview

### Solve

#### Note:

Multiple entries from [Table 349](#) must be specified in multiple `Set()` definitions, because `Set()` only takes a single option.

*Table 349 Set() in Solve{}*

<code>&lt;ident&gt;</code>	<code>=&lt;float&gt;</code>	Set node. <a href="#">Mixed-Mode Electrical Boundary Conditions on page 117</a>
<code>&lt;ident&gt;.&lt;string&gt;</code>	<code>=&lt;float&gt;</code>	Set parameter <code>&lt;string&gt;</code> of compact circuit instance <code>&lt;ident&gt;</code> . <a href="#">Mixed-Mode Electrical Boundary Conditions on page 117</a>
<code>&lt;ident&gt;</code>	<code>mode Charge</code>	Use charge boundary condition for contact <code>&lt;ident&gt;</code> . <a href="#">Changing Boundary Condition Type During Simulation on page 116</a>
<code>&lt;ident&gt;</code>	<code>mode Current</code>	Use current boundary condition for contact <code>&lt;ident&gt;</code> . <a href="#">Changing Boundary Condition Type During Simulation on page 116</a>
<code>&lt;ident&gt;</code>	<code>mode Voltage</code>	Use voltage boundary condition for contact <code>&lt;ident&gt;</code> . <a href="#">Changing Boundary Condition Type During Simulation on page 116</a>
<code>MSConfigs(</code>	<a href="#">Table 346</a>	Set MSCs. <a href="#">Manipulating MSCs During Solve on page 597</a>
<code>(PreFactor</code>	<code>=&lt;float&gt;</code> [ <code>Device=&lt;string&gt;</code> ] [ <code>MSConfig=&lt;string&gt;</code> ] [ <code>Transition=&lt;string&gt;</code> ])	Set MSC transition prefactors for emission, capture, or both using <code>EPreFactor</code> , <code>CPreFactor</code> , or <code>PreFactor</code> , respectively. <a href="#">Manipulating Transition Dynamics on page 598</a>
<code>eSHEDistributions(</code>	<code>Frozen</code> )	Freeze or unfreeze (- <code>Frozen</code> ) the electron distribution function. <a href="#">Using Spherical Harmonics Expansion Method on page 857</a>
<code>hSHEDistribution(</code>	<code>Frozen</code> )	Freeze or unfreeze (- <code>Frozen</code> ) the hole distribution function. <a href="#">Using Spherical Harmonics Expansion Method on page 857</a>
<code>HydrogenAtom</code>	<code>=&lt;string&gt;</code>	Set the concentration of hydrogen atom. <a href="#">Hydrogen Transport on page 612</a>
<code>HydrogenIon</code>	<code>=&lt;string&gt;</code>	Set the concentration of hydrogen ion. <a href="#">Hydrogen Transport on page 612</a>

## Appendix G: Command File Overview

### Solve

*Table 349 Set() in Solve{} (Continued)*

HydrogenMolecule	=<string>	Set the concentration of hydrogen molecule. <a href="#">Hydrogen Transport on page 612</a>
HydrogenSpeciesA	=<string>	Set the concentration of hydrogen atom. <a href="#">Hydrogen Transport on page 612</a>
HydrogenSpeciesB	=<string>	Set the concentration of hydrogen atom. <a href="#">Hydrogen Transport on page 612</a>
HydrogenSpeciesC	=<string>	Set the concentration of hydrogen atom. <a href="#">Hydrogen Transport on page 612</a>
TrapFilling	= <a href="#">Table 354</a>	Set trap filling. <a href="#">Explicit Trap Occupation on page 550</a>
Traps(	<a href="#">Table 354</a> )	Set traps. <a href="#">Explicit Trap Occupation on page 550</a>

*Table 350 Standalone*

Optics	Optical problem.
Wavelength	Update wavelength according to optical problem.

*Table 351 Time conditions Time-Stepping, When to Write to the Current File, When to Plot*

<float>	s A time point.
Range=(<float>*2)	Interval on time axis.
Range=(<float>*2) Intervals=<int> [Decade   Linear*]	Subdivided interval on time axis. Subdivision on logarithmic or linear scale.

*Table 352 Transient() Transient Command*

<a href="#">Table 232</a>		Time step control.
AcceptNewton Parameter		Apply relaxed Newton parameters. <a href="#">Relaxed Newton Method on page 198</a>
(	ReferenceStep=<float> )	1.e-9 s Apply relaxed Newton parameters for step size < ReferenceStep.

## Appendix G: Command File Overview

### Solve

*Table 352 Transient() Transient Command (Continued)*

BreakCriteria{	<a href="#">Table 213}</a>	Break criteria. <a href="#">Break Criteria: Conditionally Stopping the Simulation on page 119</a>
Cyclic(	<a href="#">Table 341</a> )	Cyclic analysis. <a href="#">Large-Signal Cyclic Analysis on page 145</a>
Decrement	=<float>	2 Divisor for the step size when last step failed.
Extrapolate	( <a href="#">Table 218</a> )	– Use extrapolation. <a href="#">Extrapolation on page 130</a>
FinalTime	=<float>	s Final time.
Goal{	<a href="#">Table 343}</a>	Goal for ramping.
Increment	=<float>	2 Multiplier for the step size when last step was successful.
InitialStep	=<float>	0.1 s Initial step size.
InitialTime	=<float>	0 s Start time.
MaxStep	=<float>	1 s Maximum step size.
MinStep	=<float>	0.001 s Minimum step size.
NewtonPlotStep	=<float>	Upper limit for step size for which to write Newton plot files. <a href="#">NewtonPlot on page 209</a>
NonlocalPath	( <a href="#">Table 227</a> )	<a href="#">Dynamic Nonlocal Path Band-to-Band Tunneling Model on page 529</a> , <a href="#">Handling Derivatives on page 535</a>
Plot{	Intervals=<int>	Number of intervals in Range.
	Range=(<float>*2)}	s Range of $t$ for which to generate plots. <a href="#">Saving and Plotting During a Quasistationary on page 130</a>
ReadExtrapolation		+ Try to use the extrapolation information from a previous Transient if it is available and compatible. <a href="#">Extrapolation on page 130</a>
SpectralPlot	as Plot	Plot spectral simulation results. <a href="#">Illumination Spectrum on page 651</a>

## Appendix G: Command File Overview

### Solve

*Table 352 Transient() Transient Command (Continued)*

StoreExtrapolation		+ Store the extrapolation information internally at the end of the Transient. <a href="#">Extrapolation on page 130</a>
Transient	=BE	Backward Euler method. <a href="#">Backward Euler Method on page 1199</a>
	=TRBDF	* TRBDF method. <a href="#">TRBDF Composite Method on page 1200</a>
TurningPoints(	(<float1> <float2>)...	Time point <float1> and associated advancing time-step limit <float2>.
	(Condition(Time( Table 351 [ ; Table 351]...)) Value=<float> Factor=,int>)...	Time point conditions and associated advancing time-step limit Value.

*Table 353 TrapFilling= in Set() in Solve {} Explicit Trap Occupation*

0	Set trap occupation to be in equilibrium with zero electron and hole concentration.
-Degradation	Return trap concentrations to initial values. <a href="#">Device Lifetime and Simulation on page 607</a>
Empty	(deprecated) Set all traps to empty.
Frozen	Keep the current trap occupation unchanged until the next Set or UnSet.
Full	(deprecated) Set all traps to fully occupied.
n	(deprecated) Set trap occupation to be in equilibrium with a very high electron and zero hole concentration.
p	(deprecated) Set trap occupation to be in equilibrium with a very high hole and zero electron concentration.

## Appendix G: Command File Overview

### System

Table 354 Traps() in Set() in Solve {} *Explicit Trap Occupation*

[ <string1>. ]<string2>=<float>...	Set occupation of trap <string2> of device <string1> to specified value.
[ <string1>. ]value=<float>...	Set occupation of traps of device <string1> to specified value.
Frozen	+ Freeze traps.

---

## System

Table 355 System{} *System Section*

<ident1>	<ident2> (<string>=<ident3>...)	Create device instance <ident2> from device <ident1> and connect electrode <string> to node <ident3>.
<ident1>	<ident2>(<ident3>...) {<ident4>=<pvalue>...}	Create instance <ident2> from parameter set <ident1>, connect terminals to nodes <ident3>, and override parameter <ident4> by <pvalue>, the type of which depends on the parameter.
<ident1>	<ident2>(OpticalDevice= ["<optdev1>" ...]) {<ident3>=<pvalue> ...}	Create instance <ident2> from parameter set <ident1>, connect instance to optical device instances <optdev1>, and override parameter <ident4> by <pvalue>, the type of which depends on the parameter.
ACPlot(	<a href="#">Table 356</a> )	Define circuit quantities for AC analysis output. <a href="#">Modifying Plot Output on page 109</a>
Electrical	(<ident>...)	List of electrical nodes. <a href="#">System Section on page 103</a>
HBPlot	[<string>] ( <a href="#">Table 356</a> )	Define circuit quantities for HB analysis output. <a href="#">Harmonic Balance on page 153</a>
Hint(	<ident>=<float>)	Set node only for the first solve. <a href="#">Initializing Nodes on page 108</a>
Initialize(	<ident>=<float>)	Set node until first transient. <a href="#">Initializing Nodes on page 108</a> .
Netlist	=<string>	PrimeSim HSPICE netlist file. <a href="#">PrimeSim HSPICE Netlist Files on page 95</a>

## Appendix G: Command File Overview

### System

*Table 355 System{} System Section (Continued)*

Plot	[<string>] ( <a href="#">Table 356</a> )	Plot circuit quantities to file named <string>.
Set(	<ident>=<float>	Set node. <a href="#">Initializing Nodes on page 108</a>
	<ident>.<string>=<float>)	Set parameter <string> of compact circuit instance <ident>.
SystemTopology (	Print)	Provide description and analysis of system. <a href="#">Additional Remarks on page 157</a>
Thermal	(<ident>...)	List of thermal nodes. <a href="#">System Section on page 103</a>
Unset(	<ident>)	Unset node. <a href="#">Initializing Nodes on page 108</a>

*Table 356 Plot(), ACPlot(), and HBPlot() in System{} Plotting Quantities, Modifying Plot Output, Harmonic Balance*

<ident>		Print the voltage at node <ident>.
freq	()	Print the current AC analysis frequency.
h	(<ident1> <ident2>)	Print the heat that exits device <ident1> through node <ident2>.
i	(<ident1> <ident2>)	Print the current that exits device <ident1> through node <ident2>.
p	(<ident1> <ident2>)	Print attribute <ident2> of circuit element <ident1>.
t	(<ident>)	Print the temperature at node <ident>.
	(<ident1> <ident2>)	Print the temperature difference between two given nodes.
time	()	Print the current time in transient analysis, or quasistationary t parameter for frequency-domain analysis.
v	(<ident>)	Print the voltage at node <ident>.
	(<ident1> <ident2>)	Print the voltage difference between two given nodes.

## Appendix G: Command File Overview

### TensorPlot

---

## TensorPlot

Table 357 *TensorPlot(){} Visualizing Results on Native Tensor Grid*

Name	=<string>	! Name of tensor plot. The file name for the <code>TensorPlot</code> given in the <code>File</code> section is appended by this name.
OutputLevel	=maximum	For bidirectional BPM, tensor plots are generated not only for the final solution, but also after every iteration if <code>maximum</code> is specified.
Xconst	=<float>	μm X-coordinate.
Xmax	=<float>	μm Maximum x-coordinate.
Xmin	=<float>	μm Minimum x-coordinate.
Yconst	=<float>	μm Y-coordinate.
Ymax	=<float>	μm Maximum y-coordinate.
Ymin	=<float>	μm Minimum y-coordinate.
Zconst	=<float>	μm Z-coordinate.
Zmax	=<float>	μm Maximum z-coordinate.
Zmin	=<float>	μm Minimum z-coordinate.

## Appendix G: Command File Overview

### Thermode

---

## Thermode

Table 358 *Thermode{} Boundary Conditions for Lattice Temperature*

AreaFactor	=<float>	1 Multiplier for heat fluxes. <a href="#">Reading a Structure on page 57</a>
Name	=<string>	Name of thermode.
	=Regexp(<string>)	Regular expression for matching structure contacts.
Power	=<float>	Wcm <sup>-2</sup> Heat flux boundary condition.
SurfaceConductance	=<float>	cm <sup>-2</sup> K <sup>-1</sup> W Contact thermal conductance.
SurfaceResistance	=<float>	cm <sup>2</sup> KW <sup>-1</sup> Contact thermal resistivity.
Temperature	=<float>	K Contact temperature.

---

## Various

Table 359 *Locations, all*

<a href="#">Table 360</a>		Bulk location.
<a href="#">Table 361</a>		Interface location.
Electrode	=<string>	Name of an electrode that must exist in the device.

Table 360 *Locations, bulk (dimension is that of the device)*

Material	=<string>	Name of a material.
Region	=<string>	Name of a region that must exist in the device.

## Appendix G: Command File Overview

Various

Table 361 Locations, interface (dimension is one less than that of the device)

MaterialInterface	=<string>	Material interface of the form "<ident1>/<ident2>", with <ident1> and <ident2> as the names of materials.
RegionInterface	=<string>	Region interface of the form "<ident1>/<ident2>", with <ident1> and <ident2> as the names of regions that must exist in the device and must have a common interface.

Table 362 Locations, noncontact

<a href="#">Table 360</a>	Bulk location.
<a href="#">Table 361</a>	Interface location.

Table 363 Optics() in Physics{}

<a href="#">Table 274</a>		Optics standalone.
BPMScalar(	<a href="#">Table 240</a> )	Scalar beam propagation method (BPM) solver. <a href="#">Beam Propagation Method on page 756</a>
TMM(	<a href="#">Table 240</a> )	Transfer matrix method (TMM) solver. <a href="#">Transfer Matrix Method on page 734</a>

Table 364 PMI parameters [Command File of Sentaurus Device](#)

<ident>	=<float>	Scalar floating-point parameter.
	=<string>	Scalar string parameter.
	=(<float>...)	Vector of floating-point parameters.
	=(<string>...)	Vector of string parameters.