Bell Laboratories
Lucent Technologies

subject: **Programmer's guide to the PROPHET database**

date: **November 28, 1996**

from: **Conor S. Rafferty**
**11125**
**MH 2D-303B**

ABSTRACT

This memorandum documents the database subroutines in the PROPHET simulator. The database provides access to material coefficients and tables, numerical parameters and control options in a standardized way. The database is heirarchically organized and has an inheritance facility.

## 1. Introduction

The simulator PROPHET is a computer program for the solution of the partial differential equations (PDEs) which arise in modeling semiconductor devices and processes.Any such simulator contains a number of physical models with coefficients which have been calibrated in the past or which require tuning by the simulator user. The coefficients may take the form of numbers, temperature dependent formulae, or tables of measured data. The first function of the PROPHET database is to store such coefficients. In addition it has been found convenient to store all control information for the simulator in the same database, including a description of the user's input file, a description of the set of equations to be solved, and parameters for the numerical methods. All miscellaneous information pertaining to a simulation, as opposed to structured information such as the grid and matrix, are stored in the database.

The data is organized heirarchically. At the top level, the heirarchy is structured as shown in Fig 1.

```
root {                                  % top of tree
      userinput {                       % user's parsed input file
            card1 {                     % first command
                  parm1                 % parameters of first command
                  parm2
            }
            card2...                    % second command
      }
      options {                         % control options, e.g.
            timestep                    % whether to print timesteps
      }
      library {                         % coefficient library
            physics {                   % physical coefficients
                  silicon {             % organized by material
                        ni              % e.g. intrinsic number
                        boron {         % solution quantity in material
                              Dix       % their parameters, e.g.
                        }               % diffusivity
                  }
            math {                      % numerical parameters
                  grid.ratio            % e.g. maximum size ratio of
            }                           % adjacent grid cells

            cards {                     % recognized commands
                  solve {
                        time            % parameters of "solve" command
                        temperature
                  }
            cards.defaults {
                  }
      }
}
```

Figure 1: Top level of PROPHET database.

The top level objects are the user's input file, options for this particular run, and the library. The library is divided into physical and numerical parameters, and the list of recognized commands and their defaults is also stored here. Physical parameters are organized according to material, with solution quantities appearing under the materials in which they are to be solved. The information on what equations to solve for a particular quantity is stored as a property of that quantity; e.g. the default solution method for boron is stored at `library/physics/silicon/boron/math.default`. See reference[1] for details of PDE specification.

The advantages of this storage scheme is uniform and organized access to all simulation parameters, rather than the ad-hoc common blocks and static variables of, for instance, `SUPREM4`. It also means that new modules can access user-defined parameters without making any arrangements for their storage or transmittal elsewhere in the code. Only the subroutine requesting a parameter knows of its existence. Typically a single line of code is required to request a parameter, and the request is answered by one line in the user's input file defining the value of that parameter, or one line in the stored library. The low overhead in defining new parameters is intended to discourage module programmers from hard coding coefficients or decisions, leaving as much as possible accessible outside the binary.

The following sections will describe the data structures of the database, the procedures used to access them, the i/o facilities of the database, and some higher level access functions. While not strictly a part of the database, the expression parser is also described.

## 2. Data structures

The basic object of the database is a `property`. A `property` is a typed union. The current types are real, integer, string, property list, integer array and real array (`pREAL, pINT, pSTRING, pLIST, pIARRAY, pRARRAY`) and their union names are `val.rval, val.ival, val.sval, val.lval, val.iaval, val.raval.`

A property list (`plist`) is the organizing object of the database; it contains an array of names and an array of properties. The heirarchy is constructed by putting a property list on a parent list. It is permitted (at least currently) to have multiple properties of the same name on a list.

An integer array has a dimension, a short array of bounds, and an array of integer data. A real array is the same except that the data is real. See Figure 2. The tyepdef `real` translates to `float` or `double` depending on the hardware architecture.

The `xfile` field of a property is reserved for those properties whose contents may not have been read into memory yet. This allows an integer array or real array to be inserted into the database heirarchy while postponing the possibly time-consuming input of its contents. At the time it is actually referenced, the `fill_prop` subroutine retrieves the data.

Each property list maintains a pointer to its parent, in order to support a ".." construction similar to the Unix file system. The `nalloc` field tracks the size of the currently allocated arrays of names and properties; the `nused` field tracks how many are actually being used.

The root of the heirarchy is a property pointer called `rootprop`. Its type is a property list on which the "userinput", "library" and "options" lists sit. It is initialized by the `ndb_init` call. The macro `rootlist` simplifies references to the list field in the root property.

The internal data structures of the database should not be referenced outside the database code itself. The access routines which add and delete properties and property lists are described next.

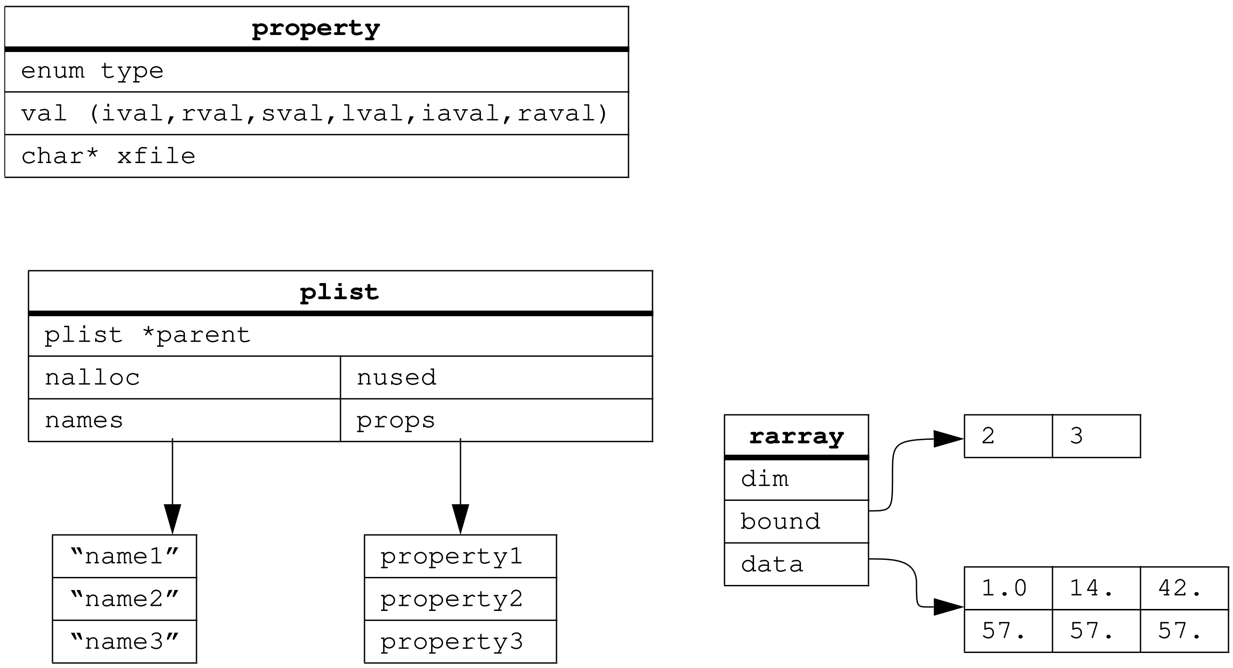


Figure 2. Data structures.

## 3. Database access subroutines.

The database has the following access subroutines, listed in table 1.

The subroutine `count_properties` returns the number of properties on a list.

The subroutine `get_local_property` returns a property by name from a property list. The property must exist in the list itself and not be inherited. A null pointer is returned if the name is not found. If there are multiple properties of the same name, the first such property is returned.

The subroutine `get_next_local_property` allows a traversal of all the properties in a list. If the `start` property is null, the first property is returned, else the next property after the given one is returned. If the `start` property does not belong to the current property list, the results are undefined.

The subroutine `get_property_by_index` returns a property by index in the list. **The index begins at one.**

The subroutine `get_property_name` returns the name of a property by index in the list. **The index begins at one.**

The subroutine `get_property` is the main access function. Given a list and a name, it looks up the name and returns a pointer to the property if it exists. If it does not exist, the list is checked for a property called `SeeAlso`. That property is a string pointing at another list through a combination of ".." and "/" similar to a Unix symbolic link. If found, that other list is then searched for the desired name. The procedure is recursive. Note that, unlike a symbolic link, a `SeeAlso` pointer

| | |
|---|---|
| int | count_properties( plist *list) |
| property* | get_local_property( char *name, plist *list) |
| property* | get_next_local_property( char* name, plist* list, property* start) |
| property* | get_property_by_index( plist* list, int index) |
| char* | get_property_name( plist* list, int index) |
| property* | get_property( char* name, plist *list) |
| property* | find_property( char* pathname, plist *list) |
| property* | fill_prop( property *p) |
| property* | put_local_property( char* name, plist* list) |
| plist* | put_local_list( char* name, plist* list) |
| int | delete_local_property( char* name, plist* list) |
| int | delete_plist( char* name, plist* list) |
| int | deep_delete_plist( char* name, plist* list) |
| int | clear_plist( plist* list) |
| int | deep_clear_plist( plist* list) |
| plist* | find_list( char* path, plist* list) |
| int | ndb_init() |
| int | ndb_check( plist* list, char *prefix) |

Table 1 - database access subroutines

always references a property **list**, not a property.

The subroutine `find_property` is similar to `get_property`, except that it takes as argument not a simple name but a pathname including any combination of "..", list names, and "/" to arrive at a valid destination. It returns a pointer to a property. It is most frequently used starting at the root of the heirarchy tree

```
find_property("library/physics/silicon/boron/Dix", rootlist)
```

to retrieve properties from the library, but can also be used to locate properties relative to a leaf property. This is the how the `SeeAlso` mechanism operates.

The subroutine `fill_prop` causes an array property to be filled from file, if it has not yet been read in. It has no effect on other properties. It returns the property being operated on, or null if there was a problem reading the array's contents.

The subroutine `put_local_property` adds a new property to a list and returns a pointer to it. The property begins life with a type of `pNONE`, and a type should be immediately assigned on return.

The subroutine `put_local_list` puts a sublist on a given list. It returns a list, that is, a pointer to the list field of the newly created property.

The subroutine `delete_local_property` removes a property from a list. It returns 0 or -1 for success or failure. If the property is a list, it can only be deleted if all its entries have been deleted.

The subroutine `clear_list` removes all the properties of a list, and returns 0 or -1.

The subroutine `delete_plist` combines the previous two functions by first clearing a property list of its properties, and then removing the list itself from its parent. It returns either 0 for success, -1 for being unable to perform the actions, or -2 if the given name exists but does not refer to a list.

The subroutine `deep_clear_plist` removes all the properties of a list, and on finding any lists, enters them and removes them also.

The subroutine `deep_delete_plist` not only clears a list and all its sublists, but removes the list itself from its parent.

To clarify, the `clear` functions remove the properties from a list; the `delete` functions remove the list itself.

The subroutine `find_list` takes a pathname and returns a list on which the requested property can be created or modified. The property need not yet exist. Although not necessary for using the routine, the mechanism by which it works is worth comment. The procedure is straightforward in the case of a property which exists; the list to which it belongs is returned. In the case of a property which exists only by virture of a `SeeAlso` pointer, a “shadow tree” is constructed (Fig. 3). The

*Before execution*

```
library/physics {
     silicon {
        boron {
           Dix = 4.2
        }
     poly {
        SeeAlso "../silicon"
     }
}
```

*After execution*

```
library/physics {
     silicon {
        boron {
           Dix = 4.2
        }
     poly {
        SeeAlso "../silicon"
        boron {
           SeeAlso "../silicon/boron"
        }
     }
}
```

Figure 3. Construction of shadow tree, before and after executing `find_list("library/physics/poly/boron/Dix")`

shadow tree has one level for each level of the real tree holding the `SeeAlso`’d property, with a link at each level to the orginal tree. The bottom list on the shadow tree is returned (that is, `library/physics/poly/boron`). This allows the user to create, in this example, a `library/physics/poly/boron/Dix` different from that of silicon, but with all other poly properties continuing to be inherited from silicon. In the case of `find_list("library/physics/silicon/boron/Dix")`, the simple list `library/physics/silicon/boron` would be returned.

The subroutine `ndb_init()` creates a root property and makes it of type `pLIST`, with 0 entries, and points the global variable `rootprop` at the root property.

The subroutine `ndb_check()` performs various database self-consistency checks, including that every property has a name and type, that array properties have valid file pointers, and that `SeeAlso` properties refer to valid destinations.

## 4. Database input/output functions

The following subroutines carry out database i/o functions.

| | |
|---|---|
| `int` | `ndberr( char* string, arg1, arg2)` |
| `int` | `ndblog( char* string, arg1, arg2)` |
| `int` | `ndbsetlog( char* name)` |
| `int` | `dump_list( plist* list, FILE* fd, int darray)` |
| `int` | `do_ndbparse( property* prop, char* filename)` |

The subroutine `ndberr` prints error messages to `stderr and` also in a log file if it has been opened. All error messages are intended to go through `ndberr`.

The subroutine `ndblog` prints run information to `stdout` and also in the log file if it has been opened. All output is intended to go through `ndbout`.

Both `ndberr` and `ndbout` resemble the `printf` function in C; the first argument is a format string and the remaining arguments are inserted into the output in printable form according to the format string. The interpretation of the string is the same as in `printf`.

The subroutine `ndbsetlog` assigns a log file to which both stdout and stderr are copied. Setting a log file allows all the error and output of a run to be collected in sequence in one file without the usual buffering sequence problems. `Stdout` is usually then redirected to /dev/null, and only `stderr` apears on the screen.

The subroutine `dump_list` dumps the contents of a list to a stream. It is useful for saving a private copy of the database after a number of user commands have modified it. The file can then be used as input to a subsequent run. The darray flag indicates whether to dump arrays raw (1 for inspection) or simply a pointer to the file from which the array came (0 for reuse in subsequent runs).

The subroutine `do_ndbparse` takes a pointer to a list property and builds a tree by reading the filename.The file format is defined by a very small `YACC` grammar; an example can be found in the appendix. It is usually used to initialize the database from file; however it can be subsequently be used to extend it by reading subtrees onto leaves of the original tree.

## 5. Expression parser

A second `YACC` grammar exists to parse arithmetic expressions. The objects on which the parser operates can be either scalars or vectors; all operations are carried out componentwise. The language allows variables to be defined for reference on subsequent calls. The parser is conveniently used for database quantities which are a function of, for instance, temperature. A first call establishes the variable `kT` in the parser's symbol table; subsequent calls can evaluate a series of expressions such as `42*exp(-3.0/kT).`

The grammar consists of the usual numerical expressions, terminated with a semicolon, and optionally preceded by "`name =`" in order to store the result in the parser's symbol table under "`name`". Exponentiation is indicated by a caret (^) and the constants `PI, E, GAMMA and DEG`

(180/π) are provided. The following functions are provided: `sin, cos, log, log10, exp, erfc, erf, sqrt, abs, normal, min, max, silcni, silceg`. Given a vector, `normal` returns a copy with all components scaled by the largest magnitude. `Min` and `max` return vectors with all components equal in size to the minimum or maximum of the given vector. `Silcni` and `silceg` are specialized functions for the intrinsic number and energy gap in silicon as a function of temperature (which have no closed form expression). The grammar also allows references to database variables through the syntax `${pathname}`; the pathname is interpreted relative to the database root property. Any mixture of vectors and reals is allowed provided all vectors have the same length; the result vector will be that length also.

The expression parser subroutines are

| | |
|---|---|
| int | vexpr( char* string, int n, real** x) |
| int | symReal( char* name, real val) |
| int | symVec( char* name, int len, real* val) |
| int | symClean( int level) |
| int | symInit() |
| int | symNode( partition* thePart) |
| int | evalcoeff( property* prop, real *value) |

The main entry point is the subroutine `vexpr`. It evaluates a string using scalars and vectors already installed in the symbol table. It returns a vector of length either one or the length of the vectors used in the expression. This vector is `malloc`'ed and should be `free`'d when no longer in use.

The subroutine `symReal` installs a named scalar in the symbol table.

The subroutine `symVec` installs a named vector in the symbol table.

The subroutine `symClean` removes all symbol table objects higher than the given level from the symbol table. Each object in the symbol table has a "level"; the lower level, the more permanent. Level 0 are the constants and functions. Level 1 is the normal variables created by assignment statements. Level 2 or higher are for variables which are intended to be of a temporary nature. Both `symReal` and `symVec` install symbols at Level 2. Thus the calls `symReal("x",1.0)` and `vexpr("x=1.0;", &n, &realp)` have similar results except that the former creates `x` with a level of 2 and the latter creates it with a level of 1. A call to `symClean(2)` would clear the former case but not the latter.

The subroutine `symInit` installs the standard constants and functions in the symbol table.

The subroutine `symNode` is a `PROPHET`-specific function which installs all the solution quantities and node coordinates as vectors in the expression parser symbol table. This can be used in calculating functions for plotting, for instance.

The subroutine `evalcoeff` evaluates a property, whether it is real, integer, string, or array, and calculates a real number. The return value is zero for success, one if given a nonexistent property, and minus one if the property existed as a string but could not be parsed. If the property is a table, it is assumed to represent a table of some quantity as a function of temperature, and the symbol `T` is retrieved from the symbol table to use as a lookup index in the property's table contents.

## 6. Higher level subroutines

A number of convenience functions are layered over the core of the database. Many exist for compatability with the first generation database. Among them are the following.

| | |
|---|---|
| int | dbase_error( int type, char* message, char* extra) |
| property* | findLib (char* name) |
| property* | findDB( char* name, int noWarning) |
| FORTRAN sub-routine | libeval( s, x) |
| FORTRAN sub-routine | dber0( s1, s2) |

The subroutine `dbase_error` is an obsolete interface to `ndberr`; the latter is preferred.

The subroutine `findLib`(s) is a shorthand for `find_property( concat ("library/", s), rootlist)`.

The subroutine `findDB` is also a shorthand for `find_property( name)`. In addition it causes a warning to be printed if the property is not found and `noWarning` argument is 0. Like `find_property`, it returns a null pointer if the property cannot be found.

The subroutine `libeval` allows library parameters to be evaluated in `FORTRAN`. It calls `find_property` on the string name and then `evalcoeff` to evaluate the property, returning a real number.

The subroutine `dber0` is a `FORTRAN` subroutine to allow `FORTRAN` output to go through the normal log mechanism (and helps avoid linking the `FORTRAN` I/O libraries!)

## 7. Summary

The `PROPHET` database provides organized access to the parameters and coefficients which govern a simulation. Subroutines to maintain and modify a heirarchy of parameters are provided, as well as an expression parser to evaluate strings stored in the database. Experience since 1990 has shown the database to be a convenient and powerful component of a modern PDE solver.

## 8. Appendix

The following is a simple database which can be read by `do_ndbparse`

```
library = (list) {
               ############################################################
               #All the cards here
               ############################################################
               cards = (list) {
                   dbase = (list) {
                       print = (int) 1;
                       printval = (int) 1;
                       printlist = (int) 1;
                       printall = (int) 1;
                       modify = (int) 1;
                       delete = (int) 1;
                       deletelist = (int) 1;
                       create = (int) 1;
                       createlist = (int) 1;
                       createdir = (int) 1;
                       deletedir = (int) 1;
                       sync = (int) 1;
                       name = (string) "";
                       file = (string) "";
                       ival = (int) 0;
                       rval = (real) 0;
                       sval = (string) "";
                       fval = (string) "";
                       from = (string) "";
                       type = (string) "";
                       list = (string) "";
                       check = (int) 1;
                       domain = (string) "";
                       temperature = (real)0;
                       pressure = (real)0;
                   };

                   caminoLoad = (list) {
                       name = (string) "";
                   };

                   field = (list) {
                       set = (string) "";
                       value = (string) "";
                   };

                   graph = (list) {
                    element= (string)"";
                    axis = (int)1;
                    boundary= (int)1;
                    materials= (int)1;
                    gridline= (int)1;
                    gridpoint= (int)1;
                    contour= (int)1;
                    cminimum= (real)0;
                    cmaximum= (real)0;
                    cdel= (real)0;
                    fill= (int)1;
                    color= (int)1;
                    yposition= (real)0;
                    xposition= (real)0;
                    itf = (string)"";
                    ymin= (real)0;
                    ymax= (real)0;
                    xmin= (real)0;
```

```
     xmax= (real)0;
     print= (int)1;
     line= (string)"";
     log = (int)0;
     star= (real)0;
     new.window= (int)1;
     outfile= (string)"";
    };

    load = (list) {
     xdr = (string)"";
     rename= (string)"";
     yscale= (real)0.0;
     list= (int) 1;
     resize.left= (real)0.0;
     resize.right= (real)0.0;
     resize.new= (real)0.0;
     switch= (string)"";
    };

    option = (list) {
     library = (string) "";
     libroot = (string) "";
     defaults= (string)"";
     version = (string) "";
    };

    solve = (list) {
     hours = (real) 0;
     minutes = (real) 0;
     seconds = (real)0;
     temperature = (real)0;
     tempfinal = (real) 0;
     cmodel = (string) "";
    };
 };

############################################################
#All the cards defaults here
############################################################
cards.defaults = (list) {

    dbase = (list) {};
    caminoLoad = (list) {};
    field = (list) {};
    graph = (list) {};
    load = (list) {};
    option = (list) {};
    solve = (list) {};
};

############################################################
# The math list
############################################################
math = (list){
 grid = (list) {
    interval.ratio= (real)1.5;
    interface.tol= (real)0.001;
    etch.overeps= (real)0.001;
    etch.eps= (real)0.001;
    expandx = (int) 0;
 };
 maxGSloop = (int) 10;
 LTE = (real) 1e-2;
 LTE.toler = (real) 1e-1;
```

```
 time.control = (string)"trbdf2";

 # All relative
 time.min = (real) 1e-8;
 time.extend = (real) 1.1;
 time.init = (real) 1e-6;

 max.color = (int) 256;
 memory.size = (int) 18;

 color-elements = (int) 1;

 noquads = (int) 0;

 plot.nx = (int) 121;
 plot.ny = (int) 101;
 plot.resolution = (real)0.005;
 plot = (list) {
    ncolors = (int)20;
    mincolor= (string)"blue";
    maxcolor= (string)"red";
    color.silicon= (string)"lightgrey";
    color.oxide= (string)"steelblue";
    color.nitride= (string)"darkgreen";
    color.poly= (string)"grey";
    color.bpsg= (string)"cornflowerblue";
    color.aluminum= (string)"navy";
    color.resist= (string)"red";
 };
};

############################################################
#The physics list
############################################################
physics = (list){
    qcharg= (real)1.602e-19;
    kboltz= (real)8.62e-5;
    celsius = (real)273.15;
    silicon= (list)
    {
     max.process.temperature= (real)1415;
     ni = (string)"silcni(T)";
     bandgap= (string)"Eg=1.17-4.73e-4*((T*T)/(T+636))";
     density= (real)2.33;
     native.oxide= (real).001;
     oxide.alpha= (real)2.2;
     spread.lateral= (real)0.66;
     default.tdel = (real)0.02;
     default.ldel = (real)0.02;
     default.thick = (real) 1.0;



     ############################################################
     # impurities for process simulation
     ############################################################
     boron = (list) {
        dsign = (int)-1;
        class = (string)"permanent";

        Dix = (string) "8.33e8/60 * exp( -3.43/kT)";
        Dip = (string) "2.5e9/60 * exp( -3.43/kT)";

        resistivity = (rarray) "Resistivity/siboron.res";
```

```
#                        Solubility = (string)"6e19*exp(-0.2/k*(1/T-1/1073))";
                         evaporate = (string)"1.674e5/60*exp(-2.481/kT)";

                         segregation-coeff.oxide.100 = (string)"2208*exp(-0.96/kT)";
                         segregation-coeff.oxide.111 = (string)"1126*exp(-0.91/kT)";
                         segregation-coeff.poly= (real) 1;
                         segregation-rate.poly= (real) 1e-3;

                         background= (real) 1e10;
                         tolerance= (real) 1e10;
                         positive= (int)1;

                      math.default = (list) {
                          implicit = (int) 1;
                          depends = (string)"psi,boron*";
                          setup = (string) "matrix.init";
                          refresh = (string) "matrix.init";
                          teardown = (string) "matrix.quit";

                          transient=(list) {
                             order=(string) "antimony,arsenic,boron,phosphorus,psi";
                             maxNewton = (int) 10;
                             topNewton = (real) 1e-2;
                             botNewton = (real) 1e-7;
                             NewtonUpd = (real) 0.05;
                             NewtonChk = (real) 1e5;
                             NewtonRhs = (real) 0.1;
                             NewtonMaxUpd= (real) 1e9;
                             NewtonAbsRhs= (int)1;
                             # All relative
                             time.min = (real) 1e-8;
                             time.extend = (real) 1.1;
                             time.init = (real) 1e-6;
                             #
                             nterm= (int)3;
                             term0= (list) {
                              geoterm = (string)"box-laplacian";
                              style = (string)"box";
                              phyterm = (string) "equilflux";
                              sol = (string)"antimony,arsenic,boron,phosphorus";
                              dep = (string)"antimony*,arsenic*,boron*,phosphorus*,psi";
                              deptype= (string) "arsenic:arsenic*=both";
                              deptype= (string) "arsenic:psi=both";
                              deptype= (string) "antimony:antimony*=both";
                              deptype= (string) "antimony:psi=both";
                              deptype= (string) "boron:boron*=both";
                              deptype= (string) "boron:psi=both";
                              deptype= (string) "phosphorus:phosphorus*=both";
                              deptype= (string) "phosphorus:psi=both";
                             };
                             term1= (list) {
                              geoterm= (string) "box-laplacian";
                              style= (string) "box";
                              phyterm= (string) "impflux";
                              sol    = (string) "psi";
                              dep    = (string) "psi";
                              deptype= (string) "psi:psi=grad";
                             };
                             term2 = (list) {
                              geoterm = (string) "diagonalweight";
                              phyterm = (string) "poissonflux";
                              sol = (string) "psi";
                         dep  = (string) "electrons,holes,arsenic*,antimony*,boron*,phosphorus*";
                              deptype = (string) "psi:electrons=conc";
                              deptype = (string) "psi:holes=conc";
```

```
         deptype = (string) "psi:arsenic*=conc";
         deptype = (string) "psi:antimony*=conc";
         deptype = (string) "psi:boron*=conc";
         deptype = (string) "psi:phosphorus*=conc";
         };
        elimination= (list) {
    order = (string) "electrons,holes,antimony*,arsenic*,boron*,phosphorus*";
         nterm = (int) 2;
         term0 = (list) {
         geoterm = (string) "diagonalweight";
         phyterm = (string) "set_active";
         sol = (string) "antimony*,arsenic*,boron*,phosphorus*";
         dep = (string) "antimony,arsenic,boron,phosphorus";
         deptype = (string) "antimony*:antimony=conc";
         deptype = (string) "boron*:boron=conc";
         deptype = (string) "phosphorus*:phosphorus=conc";
         deptype = (string) "arsenic*:arsenic=conc";
         };
         term1= (list) {
         geoterm = (string) "diagonalweight";
         phyterm = (string) "elim_carrier";
         sol = (string) "electrons,holes";
         dep = (string) "psi";
         deptype = (string) "electrons:psi=conc";
         deptype = (string) "holes:psi=conc";
         };
         };
     };
    };
 };
 arsenic = (list) {
    dsign   = (int)1;
    class   = (string)"permanent";

    Dix = (string) "3.96e8/60 *exp(-3.44/kT)";
    Dim = (string) "7.2e10/60 *exp(-4.05/kT)";

    resistivity = (rarray) "Resistivity/siarsenic.res";

    evaporate = (string) "9.0e5/60*exp(-1.99/kT)";

    segregation-coeff.oxide= (real) 800;
    segregation-coeff.poly= (real) 1;
    segregation-rate.poly= (real) 1e-3;

    background= (real) 1e10;
    tolerance= (real) 1e10;
    positive= (int) 1;
    beta= (string) "7e-66*exp(1.05/kT)";
    m   = (real) 4.0;

    math.default = (list) {
     depends = (string) "psi,arsenic*";
     SeeAlso = (string) "../../boron/math.default";
    };
 };

 phosphorus = (list) {
    dsign   = (int)1;
    class   = (string)"permanent";

    Dix = (string) "2.31e10/60 * exp(-3.66/kT)";
    Dim = (string) "2.664e10/60 *exp(-4.0/kT)";
    Dimm= (string) "2.652e11/60 *exp(-4.37/kT)";
```

```
   resistivity = (rarray) "Resistivity/siphosphorus.res";

   evaporate = (string) "9.0e5/60*exp(-1.99/kT)";

   segregation-coeff.oxide= (real) 10;
   segregation-coeff.poly= (real) 1;
   segregation-rate.poly= (real) 1e-3;

   background= (real) 1e10;
   tolerance= (real) 1e10;
   positive= (int) 1;
   beta= (string) "2.04e-41";
   m   = (real) 3.0;

   math.default = (list) {
    depends = (string) "psi,phosphorus*";
    SeeAlso = (string) "../../boron/math.default";
   };
};

antimony = (list) {
   dsign   = (int)1;
   class   = (string)"permanent";

   Dix = (string) "1.28e9/60 * exp(-3.65/kT)";
   Dim = (string) "9.0e10/60 * exp(-4.08/kT)";

   resistivity = (rarray) "Resistivity/siantimony.res";

   evaporate = (string) "9.0e5/60*exp(-1.99/kT)";

   segregation-coeff.oxide= (real) 10;
   segregation-coeff.poly= (real) 1;
   segregation-rate.poly= (real) 1e-3;

   background= (real) 1e10;
   tolerance= (real) 1e10;
   positive= (int) 1;

   math.default = (list) {
    depends = (string) "psi,antimony*";
    SeeAlso = (string) "../../boron/math.default";
   };
};

# potential & carriers
psi = (list) {
   class = (string) "permanent";
   background = (real) 0.0;
   scale = (real) 1;
   tolerance = (real) 1e-2;
   positive = (int) 0;
   Dix = (string) "11.9*8.864e-14*1e8";
   math.default = (list) {
    depends = (string) "electrons,holes";
    implicit = (int) 1;
    setup = (string) "matrix.init";
    refresh = (string) "matrix.init";
    teardown = (string) "matrix.quit";

    steady=(list) {
       SeeAlso = (string) "../../../boron/math.default/transient";
};
};
};
```

```
    electrons = (list) {
       class = (string) "permanent";
       background = (real) 1.0e-10;
       esign = (real)-1;
    };

    holes = (list) {
       class = (string) "permanent";
       background = (real) 1.0e-10;
       esign = (real) 1;
    };



    # active concentrations
# only reason that these elimination variables
    # are defined is in order to keep them around
    # afterwards for plotting and passing to device simulators.
    boron* = (list) {
       class = (string) "permanent";
       background = (real) 1.0e10;
    };
    phosphorus* = (list) {SeeAlso = (string) "../boron*";};
    antimony* = (list) {SeeAlso = (string) "../boron*";};
    arsenic* = (list) {SeeAlso = (string) "../boron*";};

   };
 };
};
```