Programmer's Reference Manual to the Kinetic Compiler version 1.00

Kenneth Geisshirt
kneth@osc.kiku.dk
Department of Theoretical Chemistry
H.C. Ørsted Institute
Universitetsparken 5
2100 København Ø
Denmark

9 October 1994

Contents

1 Introduction

This manual is the technical documentation of `kc`. The version described is 1.00. The paper contains information on the data types used in the program *i.e.*the interface is given in all details. The group of readers is thought as future programmers, who is going to extend and maintain the program.

The grammar wil also be explained. The parser and the lexical analyser will briefly discussed as well.

The reader is assumed to have knowledge of (ANSI) C, [**?**], LALR-grammars, [**?**], [**?**], [**?**], concrete data types, [**?**] and Unix in general. There exists also a user's manual to the system, [**?**], and the reader of this manual is recommended to this it before proceeding.

The manual is structured in the following way: Each section documents a concrete data type, the parser or just a module. Each section begins with a small presentation and then the operations follow, each in a distict subsection. All modules may be included more than once, *i.e.*just like the ordinary standard libraries.

The source files discussed in the manual are in general found in the `src` directory of the `kc` tree.

2 A brief overview

The kinetic compiler is a compiler in the traditional understanding, *i.e.*it translate a given source language into a target language. The source language is chemical reaction and ordinary differential equations and the target language is subroutines for a simulation program.

The general structure of `kc` is found below. The figure shows the data stream.

The lexical analyser and the parser read the input file, and they insert the information contained in the file into various tables, here called the symbol table.

A natural data type of the system is general expressions. Especially the code generators use expressions as well as the information stored by the parser. The code generators is generating the output files.

The figure below shows a simplified dependency graph of the main libraries of the program. Each library is shown as a box. The library called "Code generators" should be understood as a typical code generator.

Figure 1: The data stream in the kinetic compiler.

Figure 2: The dependency graph for the libraries.

3 Data type TableMan

The data type TableMan is found the the file `tableman.h`. The purpose of the data type is to keep track of the information, which is read from the input file. The table manager is actual seven tables in one - symbol table, reaction table, constraint table, dynamical variable table, expression table[1], print table, and parameter table - but this does the programmer not be aware of.

3.1 The `Direc` type

This type is used in some of the operations, either as input or output. The type is an enumerated type, and has three values.

Value	Description
uni	one-way reaction
bi	two-ways reaction
equi	equilibrium

These three names and the type can be used, when the table manager is included.

3.2 SetupTableMan

```
void SetupTableMan(void)
```

This procedure is setting up the table manager. It is to be called before any other routine in the table manager and only once.

This routine will always return NoError.

3.3 GetError

```
TableErrors GetError(void)
```

The function returns an error code from last used operation in the table manager. There are the following possible errors:

[1] The tables of dynamical variables and expression are closely related.

Error name	Description
NoError	no error
TooManyConst	constant table full
TooManySpec	species table full
SpecAlready	species already defined
KonstAlready	constant already defined
NonSpec	
TooManyReact	reaction table full
WrongDirect	
ReactAlready	reaction already defined
NotFound	a search was unsuccesful
TooManyConstrain	constraint table full
TooManyDynVar	dynamical table full
TooManyExpr	expression table full
ExpreAlready	expression already defined
TooManyPrn	print table full
TooManyParam	parameter table full
ParamAlready	parameter aldreay defined

The definition of these names are found in the type `TableErrors`, which can be used when the table manager is included.

3.4 NewSpecie

```
void NewSpecie(char *name, double charge)
```

The procedure defines a new species which has the name `name` and the charge `charge`. Radicals have the charge `MAXFLOAT` which is defined in standard header file `values.h`.

The possible errors are NoError, TooManySpec and SpecAlready. The last will occur, if the species already has been defined. This may mean nothing and can be ignored. The second error is more problematic; I do not have any solutions for that.

3.5 NewConstant

```
void NewConstant(char *name, double value)
```

This procedure defines a new constant in the symbol table with name `name` and the value `value`. If the constant already is defined the returned error is KonstAlready.

3.6 NewDiffConst

```
void NewDiffConst(char *name, double charge, double value)
```

The procedure assigns new value to a diffusion constant. If the species `name(charge)` has not been defined the error NonSpec is returned, otherwise NoError.

This procedure should not be used - it is an old procedure. One should use NewSpecConst instead, see section 3.42

3.7 NewCoeff

```
void NewCoeff(int react_no, char *name, double charge,
              double coeff, int side)
```

This routine assigns a new coefficient for the species `name(charge)` in reaction number `react_no`. The routine is supporting autocatalytic reactions.

If `side` is 1 the the coefficient is inserted on the left side, otherwise on the right hand side.

Please note, the procedure does *not* return any errors (not NoError neither).

3.8 NewRateConst

```
void NewRateConst(int react, int direc, Tree value)
```

The procedure sets a new value (`value`) for the rate constant in reaction `react`. The parameter `direc` gives the following possibilities.

Value	Direction
-1	$\rightarrow$
0	$=$
1	$\leftarrow$

3.9 NewBeginConc

```
void NewBeginConc(char *name, double charge, double value)
```

The NewBeginConc procedure sets a new initial concentration for species `name(charge)`. The new value is the parameter `value`. If the species is not defined, the error code is NonSpec.

3.10 NewReaction

```
void NewReaction(int react)
```

This procedure prepares a new reaction with the number `react`. There are three possible errors. NoError will be the code, if the procedure was succesful. If there was no space (reaction table full) the error code is TooManyReact, while if the reaction already has been defined, the error code is ReactAlready.

3.11 AddReactionKind

```
void AddReactionKind(int react, Direc direct)
```

This procedure adds the direction to a given reaction. Details on the directions, see section 3.1. The error code is not set in this operation, *i.e.*it can be dangerous to used it, if one is not absolutely sure on the reaction number (commonly determined by GetCurrentReact).

3.12 SpecieInReaction

```
void SpecieInReaction(int react, char *name, double charge)
```

This operation inserts a new species (`name(charge)`) into the reaction table. The species is associated with reaction number `react`. The coefficient that the species has in the reaction must be set by NewCoeff, see section 3.7. The reaction number is during parsing determined by GetCurrentReact.

There is no setting of error codes in this operation.

3.13 NewConstraint

```
void NewConstaint(char *name, double charge, Tree expr)
```

This procedure prepares a new constraint. The error code is TooManyConstrain if there is no space in the table. All constraints are assumed to be in the form

```
[J] = expr
```

where `J` is the species, *i.e.*`name(charge)`.

3.14 NumOfConstraint

```
int NumOfConstraint(void)
```

This function returns the number of constraints defined so far.

3.15 GetConstraintNo

```
void GetConstraintNo(int no, char *name, double *charge, Tree t)
```

This function finds the constraint number `no` in the constraint table. The error code is NotFound if the constraint does not exist. The species associated with the constraint is returned in the parameters `name` and `charge`.

3.16 GetReactNo

```
int GetReactNo(int counter)
```

This function returns the reaction number (as defined when it was created, *e.g.*by the parser). The parameter `counter` is the index in the table of reactions. This function is *not* pretty when used, but can be useful.

3.17 RenameSpec

```
void RenameSpec(char *rename, char *name, double charge)
```

The operation renames the species `name(charge)`, so both the name and the charge become part of the new name `rename`. The parameter `rename` has to be allocated before the call, *e.g.*by StringAlloc.

The new name will have the form `name_charge`, where the charge is converted so positive charge is n times of `p` and negative charge is n times of `n` (n is the integer part of the charge). If the charge shows that it is a radical, the suffix is `rad`.

No error codes are returned.

3.18 NoOfSpec

```
int NoOfSpec(void)
```

This function returns the number of species, which have been defined so far.

3.19 GetFirstSpecA

```
int GetFirstSpecA(int no, char *name, double *charge,
                  double *coeff, int side)
```

This function finds the first species in a in reaction `no`. If no species is found (or the reaction is not found) the function returns 0, otherwise 1. The found species

is returned in the parameters `name` and `charge`. The coefficient is returned in parameter `coeff`. The `side` parameter has be 0 if the species is to be found on the left side, otherwise 1.

This function is meant to be used together with GetNextSpecA, and GetFirstSpecA is the initial call.

3.20 GetNextSpecA

```
int GetNextSpecA(char *name, double *charge, double *coeff,
                 int side)
```

This function continues the search started by `GetFirstSpecA`, see section 3.19. The returning values are also analogous to that function.

3.21 GetCoeffInReact

```
double GetCoeffInReact(int react_no, char *name,
                       double charge, int side)
```

The function returns the coefficient (if any) of the species `name(charge)` in reaction number `react_no`. The `side` parameter determine the side of the reaction to search; 0 is left side, 1 is the right side. The function does not return any error codes.

3.22 GetFirstSpecB

```
int GetFirstSpecB(char *name, double *charge)
```

The function finds the first species in the symbol table, if there is any. If the function finds a species it returns 1, otherwise 0. The species is returned in the two parameters.

In a sense the routine (together with GetNextSpecB) is doing the same as the NoOfSpec/GetSpecNo couple.

3.23 GetNextSpecB

```
int GetNextSpecB(char *name, double *charge)
```

This function continues the search, which was begun by `GetFirstSpecB`, see section 3.22. The return values also analogue to that function.

3.24 GetSpecNo

```
void GetSpecNo(int count, char *name, double *charge)
```

The procedure finds species number `count` in the symbol table. The procedure returns the species in the two last parameters. If no species is found the return values are undefined. It is only safe to let `count` be between 1 and the number returned by `NoOfSpec`, see section 3.18.

3.25 GetReactKind

```
Direc GetReactKind(int react_no)
```

This function returns the reaction kind as defined by the type `Direc`, see section 3.1. The function has no error codes. The parameter `react_no` is the reaction number defined when the reaction was inserted into the table, and not the index of the table.

3.26 GetRateConst

```
void GetRateConst(int react_no, Direc direct, int way, Tree value)
```

The function returns the rate constant of the reaction `react_no`. The function has to known which kind of reaction it is (parameter `direct`, see section 3.1). For two-ways reactions the parameter selects which of the two constants there is returned, i.e. `way` has only meaning when `direct` = `bi`. The table below shows the possibilities.

`way`	Direction
1	$\rightarrow$
2	$\leftarrow$

3.27 GetConstant

```
double GetConstant(char *name)
```

The function returns the value of the constant `name` in the symbol table. If the constant has not been defined, the error code is NotFound, and the return value is undefined.

3.28 GetBeginConc

```
double GetBeginConc(char *name, double charge)
```

This operation finds and returns the initial concentration for the given species. If the species has not been defined, the error code is NotFound and the value of the initial concentration is undefined.

3.29 GetSpecNumber

```
int GetSpecNumber(char *name, double charge)
```

This function finds the number the species `name(charge)` has in the table. No error code is returned.

3.30 NewDynVar

```
void NewDynVar(char *name)
```

The operation inserts a new dynamical variable into the table. The error code is TooManyDynVar if there is no space for it.

3.31 NumOfDynVar

```
int NumOfDynVar(void)
```

The function returns the number of dynamical variables.

3.32 GetDynVarNo

```
void GetDynVarNo(int i, char *name)
```

The routine finds dynamical variable number `i` in the table and copy it to the variable `name`. If `i` is greater than the total number of variables the error code is NotFound.

3.33 NewExpr

```
void NewExpr(int no, Tree t)
```

This routine inserts a new expression into the expression table. The error code is TooManyExpr if there is no room in the table. The error code is ExprAlready if the expression is already defined, *i.e.*the number `no` is already used.

Expression in this context is a ordinary differentila equation. Each expression is associated with a dynamical variable, *i.e.*NewExpr is almost always used in conjunction with NewDynVar.

3.34 NumOfExpr

```
int NumOfExpr(void)
```

The function returns the number of expressions, which have been inserted into the table. No error code is set.

3.35 GetExprNo

```
void GetExprNo(int no, char *name, Tree t)
```

This function returns the expression and the associated dynamical variable inserted as number `no`. If the expression is not found, *i.e.*`no` is greater than the number of expression the error code is NotFound.

3.36 NewPowerConst

```
void NewPowerConst(int react_no, char *name, double charge, double
value, int side)
```

The routine is analogous to NewCoeff, but the variable `value` is a value for the power-law kinetics.

3.37 GetPowConstInReact

```
double GetPowConstInReact(int react_no, char *name,
                          double charge, int side)
```

The function is analogous to GetCoeffInReact, but returns the constant used in a power-law kinetics.

3.38 IsSpecInConstraint

```
int IsSpecInConstraint(char *name, double charge)
```

This function returns the constraint number, if there is any. If none found, then the function returns 0, *i.e.*the species in not constrained.

3.39 NewRateExpr

```
void NewRateExpr(int react, int direct, Tree value)
```

The routine defines a new expression for the rate of the reaction. The expression is in the variable `value`. The routine is similar to NewRateKonst.

3.40 GetRateExpr

```
void GetRateExpr(int react_no, Direc direct, int way, Tree t)
```

This function is similar to `GetRateConst`, but instead it returns an expression.

3.41 GetRateKind

```
int GetRateKind(int react_no, Direc direct, int way)
```

The function returns the kind of reaction. There are two kinds of reaction. When 2 is returned, the reaction rate is a general expression, otherwise it is more standard expressions[2].

3.42 NewSpecConst

```
void NewSpecConst(char *name1, double charge, char *name2,
                  double value)
```

Each species can have up to 10 constants associated with it. This function inserts a new one. The parameters `name1` and `charge` represent the species, while `name2` is the name of the constant and `value` is the numerical value of the constant.

The constants thought of under the development was diffusion coefficient and molar masses.

3.43 GetSpecConst

```
double GetSpecConst(char *name1, double charge, char *name2)
```

This routine retrieves what NewSpecConst inserted into the symbol table.

[2] *E.g.* law of mass action.

3.44 IsVarParameter

```
int IsVarParamter(char *name)
```

This function checks whether the variable `name` is a dynamical variable (defined by NewDynVar) or a parameter (defined by NewParameter). If `name` is a parameter, *i.e.*inserted into the expression table the return value is 1.

3.45 IsSpecParam

```
int IsSpecParam(char *name, double charge)
```

This function returns 1 if the species defined by `name` and `charge` is declared as a parameter to be used in a continuation.

3.46 NewLowHighPrefParam

```
void NewLowHighPrefParam(char *name, double low, double high,
                         double pref)
```

This function inserts informations used for continuations. If the parameter is not found, the error code is NotFound.

3.47 NewLowHighPrefConc

```
void NewLowHighPrefConc(char *name, double charge, double low,
                        double high, double pref)
```

This function is similar to NewLowHighPrefParam.

3.48 GetLowHighPrefParam

```
void GetLowHighPrefParam(char *name, double *low, double *high,
                         double *pref)
```

This procedure retrieves the information stored by NewLowHighPrefParam. If the parameter is not found, the error code is NotFound.

3.49 GetLowHighPrefConc

```
void GetLowHighPrefConc(char *name, double charge, double *low,
                        double *high, double *pref)
```

This procedure is similar to GetLowHighPrefConc.

3.50 GetInitParam

```
void GetInitParam(char *name, double *val)
```

The procedure retrieves the information stored by NewParameter, i.e. the initial value for the parameter. The error code is NotFound if the parameter `name` has not been defined.

3.51 GetDeltaParam

```
void GetDeltaParam(char *name, double *val)
```

This routine is similar to GetInitParam, but it retrieves the initial step length for the parameter.

3.52 GetDeltaConc

```
void GetDeltaConc(char *name, double charge, double *val)
```

This routine is similar to GetDeltaParam.

3.53 GetCurrentReaction

```
int GetCurrentReaction(void)
```

The function returns the number of the reaction being parsed.

3.54 NoOfReact

```
int NoOfReact(void)
```

The return value is the number of reactions parsed. Notice that bidirectional reactions count only as one reaction!

3.55 SumCoeff

```
double SumCoeff(int react_no, int side)
```

The routine sums up the coefficients in reaction `react_no`. The argument `side` should be 1 if the summing should be the left-hand side, otherwise it will be the right-hand side.

3.56 IsSpecInReact

```
int IsSpecInReact(int react_no, char *name, double charge,
                  double *coeff)
```

The function returns 1 if the species `name(charge)` is found in reaction `react_no`. The total coeffient is also returned.

3.57 NewParameter

```
void NewParameter(char *name. double init_val)
```

The routine declares a new parameter (used in continuations) with the name `name`. The initial value of the parameter is given by the parameter `init_val`.

The error code is ParamAlready is the parameter has already been declared, TooManyParam indicates there is no space for the parameter, and NoError indicates that the call was succesful.

3.58 NewDeltaParam

```
void NewDeltaParam(char *name, double delta)
```

The routine stores a new step size for the parameter `name`.

3.59 NewDeltaConc

```
void NewDeltaConc(char * name, double charge, double delta)
```

Similar to NewDeltaParam, but the parameter is given by `name` and `charge`.

3.60 NewParamConc

```
void NewParamConc(char *name, double charge, double init_val)
```

Similar to NewParam, but the parameter is given by `name` and `charge`.

3.61 NumOfParameter

```
int NumOfParameter(void)
```

The function returns the number of continuation parameters declared so far.

3.62 GetParamNo

```
void GetParamNo(int no, char *name, double *charge, int *form)
```

The routine finds parameter number `no`. If `form` is 2, then the parameter is a species (and `charge` is used), 1 indicates a ordinary parameter, and 0 an error.

4 Data type SymbMath

The concrete data type SymbMath is capable of handling expressions in a symbolic way. The library is defined by the file `symbmath.h`.

The goal has been to create a general-purpose expression handler, *i.e.*a library which can do the common mathematical manipulations. Common manipulutions are basic operators (*e.g.*addition), functions (*e.g.*sin), and differentiation.

The expressions are implemented as binary tree. Some simplifications are also done, but these are "invisible" to the user, *i.e.*they are called implicitly.

Expressions (or trees) have to be declared by the user. Let `t` be the name of the tree, which have to be declared. The declaration `Tree t;` will be sufficient. Before the use of the tree, the tree has to be created or allocated, see section 4.2.

The library handles two kind of "values". They are constants and variables. Constants are just floating-point numbers (of the type `double`). The variables are strings of characters. Note, that there is no check wheather the characters are printable or not.

4.1 TreeGetError

```
int TreeGetError(void)
```

This is the error handler routine of the library. The error codes are defined as macros, and they can be used when the SymbMath library is included. The following error codes are defined:

Error	Description
NoError	no error
NoEval	could not evaluate tree
NoTree	no tree allocated

4.2 TreeCreate

```
Tree TreeCreate(void)
```

This operation creates a tree, *i.e.*allocates the right portion of memory and returns a pointer to it. The operation always leave a NoError code.

4.3 TreeAdd, TreeSub, TreeMul, TreeDiv, and TreePow

```
void TreeAdd(Tree t1, Tree t2)
```

```
void TreeSub(Tree t1, Tree t2)
void TreeMul(Tree t1, Tree t2)
void TreeDiv(Tree t1, Tree t2)
void TreePow(Tree t1, Tree t2)
```

These five operations perform the basic five arithmetic operations, *i.e.*$t1 = t1 opt 2$. The error code is always set to NoError.

4.4 TreeSign

```
void TreeSign(Tree t)
```

This operation changes the sign of the expression given by t, *i.e.*$-t$.

4.5 TreeAssignConst

```
void TreeAssignConst(Tree t, double val)
```

This function sets a tree equal to a constant (`val`). The function semantic is much like $t =$ `val`. The function always returns NoError.

4.6 TreeAssignVar

```
void TreeAssignVar(Tree t, char *name)
```

The function is similar to `TreeAssignConst`, see section 4.5. Instead of a constant, the tree is assigned to a variable, *i.e.*the semantic is $t =$ `name`.

4.7 TreeSubstVar

```
void TreeSubstVar(Tree t, char *name, double val)
```

The function substitutes all occurences of the variable `name` in the tree `t` with the value `val`. If the variable is not in the tree, the tree is not changed.

4.8 TreeDerive

```
void TreeDerive(Tree res, Tree t, char *name)
```

This operation differentiates the expression with respect of the variable `name`. The result of the differentiation is returned in `res`. The function will always return NoError as error code.

4.9 TreeEval

```
double TreeEval(Tree t)
```

This function tries to evaluate the expression `t`, *i.e.*simplify it to a constant. If it is not possible, then the error code is NoEval, otherwise NoError. If it was not possible (a variable is in the tree), the returned value is undefined.

4.10 TreePrint

```
void TreePrint(Tree t, int mode, FILE *output)
```

This routine prints the tree `t` to the file `output`. The tree is printed to the screen, if `output` is set to `stdout`. The parameter `mode` determines how the output is going to look like. At the moment three modes are supported.

mode	Description
1	Fortran-77
2	Pascal
3	ANSI C

4.11 TreeCpy

```
void TreeCpy(Tree t, Tree res)
```

This operation makes an exact copy of `t` and places it in `res`, *i.e.*the operation is `res` = `t`

4.12 TreeKill

```
void TreeKill(Tree t)
```

This is the opposite of `TreeCreate`. The operation deallocates a tree.

4.13 TreeSubstTree

```
void TreeSubstTree(Tree t, char *name,
                   Tree value)
```

This routine is analogue to `TreeSubstVar` but instead of a value an expression is substituted.

4.14 TreeApplyFunc

```
void TreeApplyFunc(Tree *t, Function func)
```

This function applies a given function to the expression hold by `t`. The functions available are: `Exp`, `Sin`, `Cos`, `Tan`, `Ln`, `Log`, `Cosh`, `Sinh`, `Tanh`, `Asin`, `Acos`, `Atan`, `Acosh`, `Asinh`, and `Atanh`.

5 Module CodeCall

This module is implemented by two files, `codecall.h` and `codecall.c`. There is only one procedure in the module and it is CodeGenCall. It has the function head:

```
void CodeGenCall(int mode)
```

The implementation of CodeGenCall includes *all* code generators. Each code generator has its one file, which makes it easy to organise. The parameter `mode` determines which code generator is called. Pseudo code of the function is:

```
case mode of
 1 : call code generator 1
 ...
 n : call code generator n
```

Before calling the code generator there will be opened the files which the generator is going to use. But this can the done otherwise (let the code generator open the files).

6 Grammar, semantic action, *etc.*

In this section I will explain the grammar, the parser and the lexical analyser. If a future programmer will charge anything in these three parts, he (or she) is asked to contact me first.

6.1 The Grammar

The grammar is a LALR(1)-grammar. That means that programs like *yacc* can generate a parser directly from it. The grammar is made so much left-recursive as possible.

The parser is found in `kc.y` while the lexical analyser is found in `kc.l`.

6.2 The parser

The parser is generated by `yacc` directly from the grammar. A good and general book on `yacc` is [**?**]. There are inserted semantic actions into the grammar.

Not all the semantic actions are using the parser stack. The number of actions not using the stack is minimal. They are using local variables instead. These variables are:

Variable	Type	Function
`name`	`char *`	storage of strings, misc. names
`charge`	`double`	charge of species
`coeff`	`double`	coefficient in reaction
`flag`	`char`	flag, used in various situations
`lineno`	`int`	contains the line number in the input

The parser stack is declared by a union in the file containing the grammar and the semantic actions. This union contains the following fields:

Field	Type	Description
`dval`	`double`	misc. floating-point values
`oper`	`char`	operators in expressions
`name`	`char *`	misc. names read by the lexical analyser
`flag`	`int`	flag, used in various situations
`compound`	`comp`	a compound[3]
`tree`	`Tree`	expression
`func`	`Function`	function

It is recommended that the programmer uses the parser stack (*i.e.*the union) and not some global variables.

6.3 The lexical analyser

The lexical analyser is generated by `lex`. Almost all actions are just returning a token value.

7 Code generation

When all input have been parsed and stored in the various tables, the code generator produces the output (the code). But since almost any code generator share some common code, a special module has been written. The name of the module is `codegen.c`.

The module declares a number of variables, namely

Name	Description
v	rate expressions
con	constraints
jacobi	jacobian matrix (1st derivatives)
hess	the hessian tensor (2nd derivatives)
keld	3rd derivatives

They are all defined as arrays or arrays of arrays of tree, see section 4.

7.1 InitCodeGenVar

```
void InitCodeGenVar(int n, int m)
```

This is the first routine to be called by a code generator. The routine allocates space for the variables discussed in section 7. The argument `n` is the number of dynamical variables and `m` is the number of constraints.

7.2 GenerateRateExpr

```
void GenerateRateExpr(int mode, int ngrid, int mgrid, int boundary)
```

This function calculates all the rate expressions and constraints. If `mode` is 1, it is ordinary kinetics, while 2 is a reaction-diffusion system. The arguments `ngrid` and `mgrid` is the number of grid points for the reaction-diffusion system. The last argument `boundary` is 1 if no-flux and 2 if periodic boundary conditions.

7.3 GenerateJacobi

```
void GenerateJacobi(int mode, int ngrid)
```

This routine finds the Jacobian matrix from the rate expressions. The parameters are the same as for GenerateRateExpr, see subsection 7.2.

7.4 GenerateHessian

`void GenerateHessian(void)`

This routine is computing the elements of the hessian tensor, *i.e.*

$$H_{ijk} = \frac{\partial^2 f_i}{\partial x_j \partial x_k}.$$

7.5 GenerateKeldian

`void GenerateKeldian(void)`

The routine is computing the elements of the tensor:

$$K_{ijkl} = \frac{\partial^3 f_i}{\partial x_j \partial x_k \partial x_l}.$$

8 Code generators

This section documents the code generators already in action and how to write a new one. I will claim that is not difficult to write a new one, and I will give some hints.

8.1 Writing new code generators

The most obvious extension of the program is properly new code generators. This section will describe how to write one. Additional information is found in section 7.

The code generator is the back-end of the system. It is the final step in transforming the input into the desired output. After the parsing, the chemical model is put into various tables. The way to get the information out of the tables is defined in section 3. During the code generation some symbolic manipulation of expression can be needed, see section 4.

Examples of code generators can be found in the files `kgode.c` and `finn.c`.

The exists some common constructions in every code generator, which I will show below. Beside them, I have written a few routines which do some common work. They are described in section 7.

Almost all code generators will have some construction in common. I will show them and give some possible solution.

The first construction is "for all reaction". This can be made by

```
for(i=1;i<=NoOfReact();i++) {
  ...
}
```

Any reference to the reaction is done directly the i, *e.g.*`GetReactNo(i-1)`.

Similar to the previous, the construction "for all species" can be made by

```
for(i=1;i<=NoOfSpec();i++) {
  ...
}
```

There is one remark about this approach. If the code generator is going to use the species (i.e. species number i) a construction like `GetSpecNo(i, ...)` is appropriate.

8.2 KGode

```
void KGode(FILE *ccode, FILE *hcode, int mode)
```

This code generator is the largest and the most used by many users. It generates code for simulating chemical reactions and solving ordinary differential equations.

The two file handlers (`ccode` and `hcode`) points to the files `model.c` and `model.h`.

The meaning of the `mode` parameter is found in the table below.

Value	Description
1	Ordinary differential equations and chemical kinetics
2	Reaction-diffusion equations
3	Shifting between sets of equations

Mode 3 uses a number of internal routines, and they are documented below. These routines are operations on a data type called a base name table which keeps track of the different sets of equations.

The records of a base name table is simply a name and pointers (implemented as indices) to the rate expressions.

8.2.1 StripName

```
void StripName(char *name, int *part)
```

A name of dynamical variable consists of a name followed by a number. The name is called the base name. The number shows which set of equation, the actual differential equation is part of. StripName determines the base name and which set of equations it belongs to. The argument `name` is both input and output, and `part` is zero, if no base name fits.

8.2.2 BuildBaseTable

```
void BuildBaseTable(void)
```

The base name table is build up by this routine.

8.2.3 GetIndex

```
int GetIndex(char *name, int part)
```

The routine returns an index to the rate expression array (the global variable `v`, see section 7), where the dynamical variable `name` in set `part` is found.

8.2.4 NumOfBaseNames

```
int NumOfBaseNames(void)
```

The function returns the number of base names stored in the base name table. It is very useful in loops.

8.2.5 GetBaseNameNo

```
void GetBaseNameNo(int i, char *name)
```

The routine copies the base name number `i` into `name` as found in the base name table.

8.3 Finn

```
void Finn(void)
```

This code generator is an example of a code generator which does not generate code. On the other hand it computes the jacobian matrix numerically and calcutes the eigenvectors and eigenvalues.

The code generator is heavily using some numerical libraries, namely `eigen`, `complex` and `matrix`. They are documented in [**?**].

8.4 KNcont

```
void KNcont(FILE *code)
```

The code generator is used together with Keld Nielsen's continuation program written in Pascal.

8.5 Waves

```
void waves(FILE *hcode, FILE *ccode, FILE *icode)
```

The code generator is used together with Kenneth Geisshirt's simulation programs for reaction-diffusion systems.

9 Module Misc

I have written a small module called Misc. The module is defined by the include file `misc.h`. The module consists of a number of routines which do not fit into other modules.

9.1 GetAndPrintConst

```
void GetAndPrintConst(char *name, char *text, int type
     double def, FILE *output, int mode)
```

The procedure find the value of the constant `name`, prints an assignment statement on file `output` of the form: `text` assignment-operator value. The assignment-operator depends on the `mode`: Fortran (1), Pascal (2), and C (3). If the constant has not been defined a default value is used (`def`). The procedure will also print the appropriate line seperator character according to the mode.

The routine is *very* useful when one wants to print a number of constants in a code generator, *i.e.*it is used to generate the initialising code for a simulation program.

9.2 Fact

```
int Fact(int n)
```

This is simply just the factorial, *i.e.*$n!$.

9.3 StringAlloc

```
char *StringAlloc(void)
```

The routine allocates space for a string of a given length (see the file `config.h`).

9.4 StringFree

```
void StringFree(char *str)
```

The routine frees the space used by the string `str`.

10 Advices and hints

This section gives some advices and hints on the work with `kc`. The work is seen from the programmer's view and not the user's.

It should be noted that the program is written in ANSI C. This may course trouble on systems without a ANSI-C compiler (old systems may have only a K&R-C).

The installation is very simple for many platforms. There is a script called `kc-inst` which does the work. Run the script without any arguments to get some help.

The package is fairly easy to port. I have it running on HP-UX (Hewlett-Packard), Linux (Intel based computers), MS-DOS (Intel based computers), ConvexOS, IRIX (Silicon Graphics), and Ultrix (Digital). With a standard C-compiler like `gcc`, there should be no problems.

The program is configurated in `config.h`. A number of macros is defined, and the table below gives a short introduction to them.

Macro	Description
VERSION	A string giving the version number.
STRING_LENGTH	The length of strings used.
MALLOCTYPE	The type used by the standard function `free`.

The macro `_PLATFORM_*` is usually set up in the makefile, and it gives which platform (operating system, compiler, *etc.*) being used.