AVS CHEMISTRY DEVELOPER'S GUIDE

Release 4.0
June, 1992

Advanced Visual Systems Inc.

Part Number: 320-0017-02 Rev A

NOTICE

Advanced Visual Systems Inc.
300 Fifth Ave.
Waltham, MA 02154

TABLE OF CONTENTS

4 CDK Programmer Notes

5 CDK Library Reference

6 CDK Examples

7 CDK Module Man Pages

CHAPTER 1

CDK OVERVIEW

Introduction

In order to better address the needs of the chemistry community, a **Molecule Data Type** (**MDT**) has been added to AVS. The MDT addresses the general needs of classical, substructure, and quantum chemistry fields. It is contained in releases from AVS 3 on.

The **Chemistry Developers Kit** (**CDK**) consists of a set of Fortran and C libraries to facilitate use of the Molecule Data Type, and a collection of modules, networks, and test data to demonstrate its use.

This manual describes what a programmer needs to know to write an AVS chemistry module using the Molecule Data Type. The manual assumes an elementary understanding of the concept of a data flow network and a working knowledge of either the C or the Fortran programming languages for writing modules. For AVS user documentation, see the *AVS User's Guide*, and the *AVS Developer's Guide*.

The CDK is a product that provides the tools necessary for end users, Value Added Resellers (VARs), Independent Software Vendors (ISVs) and others, to build chemistry applications on top of AVS. These applications, developed in the AVS framework, may then be run on other platforms licensed for AVS. All AVS licensed platforms support the Molecule Data Type.

CDK Concepts

Creating chemistry modules using the Chemistry Developers Kit is a relatively simple task. Although the library, *libchem*, is large it is conceptually simple to grasp. Central to an understanding of the CDK is a firm grasp of the Molecule Data Type. This data type is composed of 10 hierarchically arranged CHEM"object"s that may be grouped together in linked lists. They may be used as building blocks to best represent the application's needs. Detailed information on the CDK data types is presented in Chapter Two of this manual.

The CHEM"object"s are central in understanding the Application Programmers Interface for the CDK. Each CHEM"object" has a complete set of func-

tions that allow access to all elements of that object; in addition there is a complete set of functions to manipulate lists of these objects. Both of these classes of functions, accessor and list manipulation, use standardized naming conventions. A third class of function, general_utility, provides translation and management tools for the CHEM"object"s. Prior to searching out explicit function names, it is suggested that you review the "Nomenclature" chapter, Chapter Three of this manual. In addition, the "CDK List Manipulation Functions" section of Chapter Five will provide additional information about the list manipulation functions.

The "Programmers' Notes", Chapter Four of this manual, lists numerous hints helpful in programming with the CDK libraries. It should be reviewed before starting extensive programming endevours.

Creating Modules with the CDK

The field of chemistry has large number of data formats in use. While this manual cannot specifically address them all, it does attempt to describe the process of translating foreign data into the AVS supported Molecule Data Type (MDT). The most common type of chemistry data comes in blocks of assorted fixed length records. This section examines an example of this format, and the process necessary to translate it into its MDT counterpart.

The CDK comes complete with example reader and writer modules which understand and communicate a data file format known as a **structure** file. A structure file consists of ***.con*** and ***.fch*** files which contain the description of a molecule in fixed length records. The structure files serve only as examples of a type of chemistry data and should not be mistaken for an approved CDK file format. An example of a structure file follows:

- The first line contains the following fields:
  - Atom count, a 3-digit integer.
  - Structure title, up to 69-characters.
- The subsequent lines contain the following fields:
  - Atom label, 2-characters.
  - Atom index, a 5-digit integer.
  - X location, a 12-digit real (f12.6).
  - Y location, a 12-digit real (f12.6).
  - Z location, a 12-digit real (f12.6).
  - MM2 type, a 5-digit integer.
  - Connectivity, six 5-digit integers.

The example C reader module, */usr/avs/examples/chemistry/CHEMcon_r.c,* requires that the fields described above be separated by blanks. This is the same as the supplied **Read structure file** module.

The example Fortran reader module, */usr/avs/examples/chemistry/CHEMcon_r-f.f,* uses two formatted Fortran read statement as follows:

```
format(i3,a69)
format(1x,a1,a1,i5,3f12.6,i5,6i5)
```

The associated *.fch*, or "formal charge file", contains records with:

- Atom index, a 5-digit integer.
- Formal charge, a 10-digit real (f10.6)

The C example uses the atom index; the Fortran sample program skips over it (5x,f10.6).

The *h2co.con* and *h2co.fch* files, provided with this release in the directory */usr/avs/data/chemistry*, are depicted below:

```
  h2co.con
-------------------------------------------------------------------------------
  4 H2CO - AVS test
  C    1   -0.047600   -0.023100    0.033000    3    4    3    2    0    0    0
  O    2    1.158600   -0.134100    0.145700    7    1    0    0    0    0    0
  H    3   -0.584200    0.939000    0.131900    5    1    0    0    0    0    0
  H    4   -0.722000   -0.873800   -0.179100    5    1    0    0    0    0    0


  h2co.fch
-------------------------------------------------------------------------------
    1  0.292100
    2 -0.290200
    3 -0.001000
    4 -0.001000
```

In general, chemistry data is well suited for representation in a hierarchical format. For example, to create a description of a classical molecule, we may arrange the molecule h2c0 accordingly:

```
CHEMmolecule 1: h2c0
     CHEMatom 1:C
          CHEMcandb 1: CHEMatom 2, Double Bond
          CHEMcandb 2: CHEMatom 3, Single Bond
          CHEMcandb 3: CHEMatom 4, Single Bond
     CHEMatom 2: O
          CHEMcandb 1: CHEMatom 1, Double Bond
     CHEMatom 3: H
          CHEMcandb 1: CHEMatom 1, Single Bond
     CHEMatom 4: H
          CHEMcandb 1: CHEMatom 1, Single Bond
```

In this example the pseudo CHEM"object"s, CHEMmolecule, CHEMatom and CHEMcandb represent the molecule, atom, and connectivity/bond data for h2c0. Each of these CHEM"object"s is in itself a list. CHEMatoms 1, 2, 3 and 4 are contained in a list that comprises the atoms for CHEMmolecule h2c0. Note that there are several lists with only one member.

In order to use AVS for chemistry applications, the original data must be transformed into the Molecule Data Type format. The process generally in-

volves reading the existing data into memory and then parsing it to create valid CHEM"object"s. Ideal C and Fortran examples exist on-line in the files */usr/avs/examples/chemistry/CHEMcon_r.c* and *CHEMcon_rf.f*. Printouts of these examples are in the "Examples" chapter. Review of these modules will provide explicit examples of creating valid CHEMmolecule objects.

CHAPTER 2

CDK DATA TYPES

Introduction

The Molecule Data Type (MDT), is composed of 10 chemistry building blocks, or CHEM"object"s, that can be used in various permutations to address the needs of the application. In the Network Editor the Molecule Data Type is identified by the color magenta. The MDT promotes software reuse by defining a set of common chemistry data types for application and module writers to use. The application is responsible for determining if the molecule data received will fulfill its needs.

All functions and subroutines are derived from the data structure names and elements. The MDT can represent one or more molecules and address any combination of classical, substructure, or quantum representations. Each CHEM"object" is an aggregate type. For example, CHEMmolecule is composed of other CHEM"object"s and primary types. C programmers will observe that a CHEM"object" is physically a structure; while Fortran programmers need only perceive them as symbolic units. The Fortran interface library, *libchem_f*, manages all pointer and interlanguage calling issues. It should be noted that CHEM"object"s are not object oriented.

Programming using CDK involves creating instances of the required CHEM"object" and assigning data to elements of that instance. In the case where a CHEM"object" element refers to another CHEM"object" list, the CHEM"object"s must adhere to the defined MDT hierarchy.

A number of the CHEM"object"s contain a name element. The names are dynamic character strings that grow or shrink upon demand; they are managed internally by the CDK. Both Fortran and C programmers can take advantage of this feature.

A number of the CHEM"objects" have references to the User Data type defined in the "Advanced Topics" chapter of the *AVS Developer's Guide.*

Each of these CHEM"object"s can be a member of a homogeneous linked list. Lists can be constructed and used as hierarchical building blocks to address the application's data structures requirements. Linked lists were adopted to ensure that there are no software limits on the number of allowable CHEM"object"s and to keep communication overhead down. All CHEM"object" lists are one-directional lists.

The following figure shows a list of CHEMmolecules. One instance of CHEMmolecule exhibits its CHEMatom list and an associated connectivity and bond (CHEMcandb) list. There are other CHEM"object"s that are not included in this diagram.

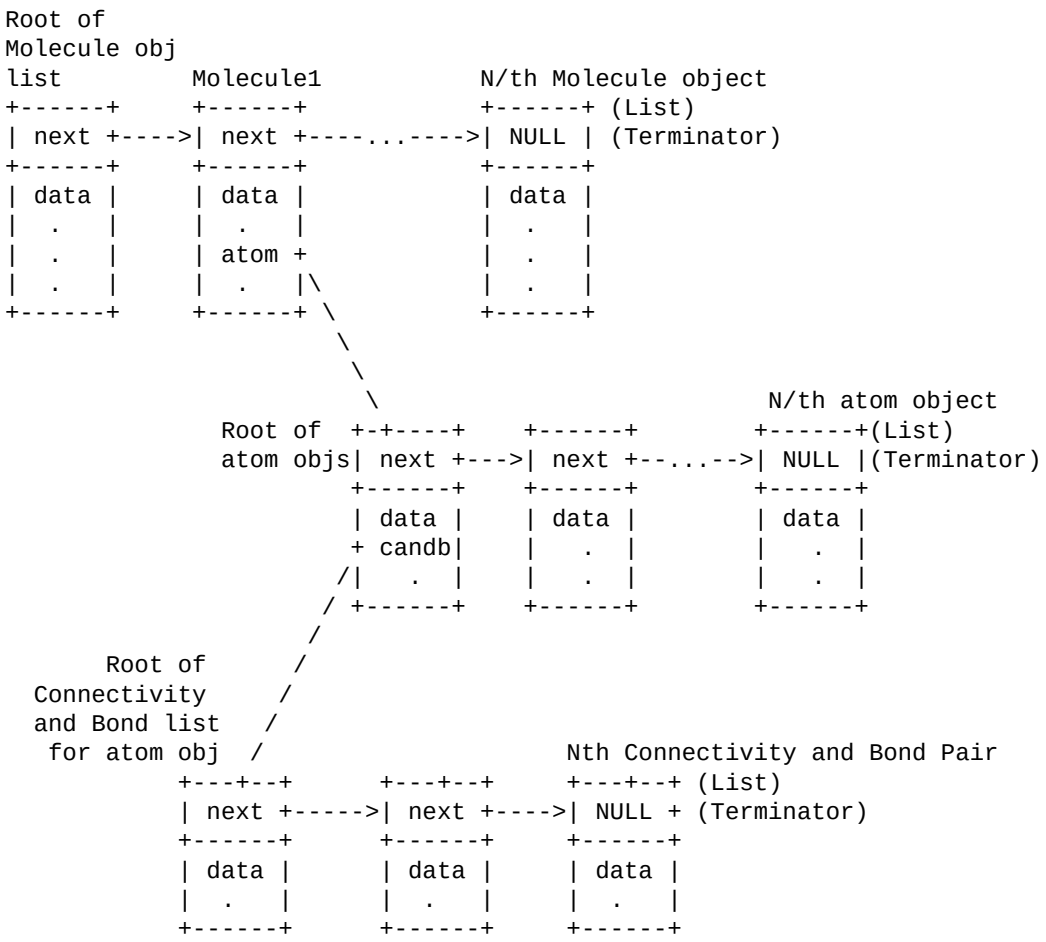


Figure 2-1 An Example MDT Linked List

Hierarchy

The root of the MDT hierarchy is the CHEMmolecule object; it is the data type registered for passing between modules in the Network Editor. From this root other CHEM"object"s branch and facilitate the representation of classical, quantum and substructure data. The full tree of CHEM"object"s is depicted in Figure 2-2, below.

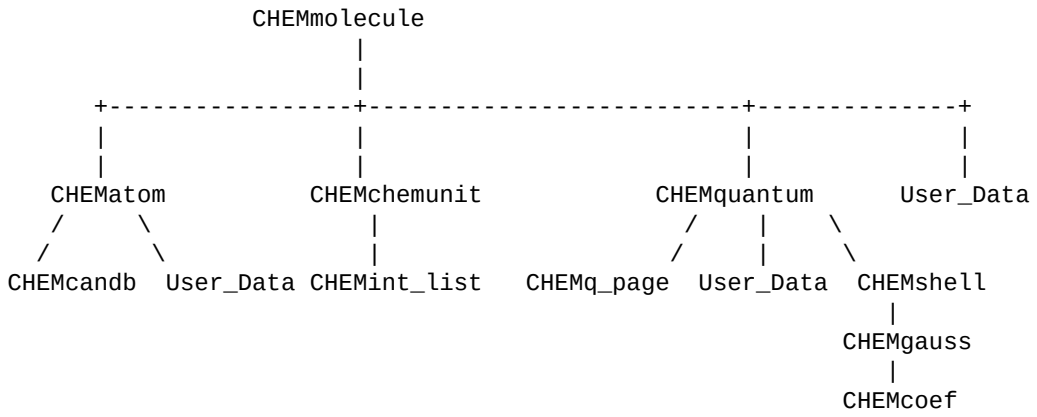


Figure 2-2 CHEM"object" Hierarchy

For a given application, one need only create and manipulate those CHEM"object"s of interest. This allows the construction of modules that focus on the specifics of the discipline and helps keep the communication overhead at a minimum. The order of the hierarchy should be strictly adhered to. It is not acceptable, for example, to attach CHEMatoms directly to CHEMchemunits.

The CHEM"object"s CHEMmolecule, CHEMatom, and CHEMquantum have instances of User_Data. Each of these instances can be used to represent user-specific needs. The CHEMatom object has an internal representation of User_Data that can be used to extend the number of CHEMatom elements. It can be replaced with another or external User_Data definition. The application is responsible for the creation and management of any external User_Data structures.

CDK Data Types

The following CHEM"object"s comprise the MDT. Each of these objects must be created with the appropriate **CHEM"object"_alloc** function. They must also be deleted with the appropriate **CHEM"object"_del** or **CHEM"object"_free** function. The delete function will resolve any references in a CHEM"object" list to the object to be deleted, and the free function will not.

The term "level" in the following descriptions is used to refer to the depth of the CHEM"object" in the MDT tree.

CHEMmolecule Object

The CHEMmolecule object is the root object of any AVS Molecule Data Type structure. All second level CHEM"object"s must refer exclusively to one instance of CHEMmolecule. It is not acceptable to have any ancillary

CHEM"object"s of CHEMmolecule A referred to, or by, CHEMmolecule B or any of its CHEM"object"s. The CHEMmolecule can contain any combination of CHEMatom, CHEMchemunit, or CHEMquantum lists. These CHEM"object"s facilitate the representation of classical, sub-structure, and quantum mechanical data. The User_Data node can be extended to address application needs. However, it is up to the application to manage and validate its representation.

CHEMmolecule is composed of the following elements:

```
char               *name;   molecule name
int                units;   units: ANGSTROMS or BOHRS
int                natom;   number of CHEMatoms
int              nc_unit;   number of CHEMchemunits
int               nquant;   number of CHEMquantums
CHEMatom           *atom;   pointer to CHEMatom object list
CHEMchemunit     *c_unit;   pointer to CHEMchem_unit object list
CHEMquantum       *quant;   pointer to CHEMquantum object list
char          *User_Data;   User Data node
```

CHEMatom Object

CHEMatom is a second level CHEM"object". A CHEMmolecule object can refer to one or more CHEMatoms. The CHEMatom object facilitates a classical atom representation. It can be used in conjunction with a CHEMquantum object, if desired, and it is necessary in defining valid CHEMchemunit objects. The CHEMcandb object list can be defined if connectivity and bond information is required. In addition, the User Data element allows the application to extend the CHEMatom object definition. However it is up to the application to manage and validate the User_Data representation.

CHEMatom is composed of the following elements:

```
int         index_number;    number as assigned by
                             the application
char               *name;    name
int                color;    color
double           x, y, z;    position
float             radius;    radius
CHEMcandb           *cnb;    pointer to CHEMcandb object list
char          *User_Data;    User Data node
```

CHEMcandb Object

CHEMcandb is a third level CHEM"object". A CHEMatom can refer to a list of one or more CHEMcandbs. The CHEMcandb list represents connectivity and bond information for a given atom. CHEMcandb objects can represent either symmetric or asymmetric representations of CHEMatom connectivity.

The symmetrical model requires that each CHEMcandb object represent half a bond. The bond starts at the location of the parent CHEMatom and continues to the mid point of the line that defines the connectivity for the two CHEMatoms. If CHEMatom A has a double bond to CHEMatom B, for example, then CHEMatoms A and B must each have instances of CHEMcandb that reference each other. Each CHEMcandb should both have the same bond type.

The asymmetric model requires that each instance of CHEMcandb represent an entire bond. The bond starts at the location of the parent CHEMatom and terminates at the designated CHEMatom. For example, if CHEMatom A has a double bond to CHEMatom B, then CHEMatom A need only have an instance of CHEMcandb that references CHEMatom B. The connectivity should not be reciprocated by CHEMatom B. Thus, it is possible to have CHEMatoms that have no locally associated connectivity, yet are properly connected: their connectivity is defined by some other CHEMatom instance.

The CDK modules provided expect symmetric connectivity. There are functions to aid in the conversion from the symmetric to asymmetric representations and vice versa.

The CHEMcandb valid bond types include

- **CHEM_BOND_NONE** - null or undefined bond,
- **CHEM_BOND_SINGLE** - single bond,
- **CHEM_BOND_DOUBLE** - double bond,
- **CHEM_BOND_TRIPLE** - triple bond,
- **CHEM_BOND_HYDROGEN** - hydrogen bond,
- **CHEM_BOND_DISULFIDE** - disulfide bond, and
- **CHEM_BOND_AROMATIC** - aromatic bond.

CHEMcandb contains the following elements:

```
int             conn;      internal atom offsets 0 to (natoms-1)
int               bond;      bond types.
```

CHEMatom Internal User Data

Each CHEMatom has an instance of User Data. The application can choose between an internally defined instance of User Data or define its own external version. The internal CHEMatom User Data structure is only valid for CHEMatom objects and effectively extends the CHEMatom definition. The internal definition of User Data addresses general needs of a classical atom structure. The user can initialize the use of this data structure in a module via a call to **CHEMatom_init_User_Data()**. Further information is available in the "CDK Library Reference" chapter under the "CHEMatom" section in this manual. The application is responsibile for the management and validation of any external User_Data representation.

The internal definition of CHEMatom User_Data contains:

```
int             parent;     offset of parent molecule
int              a_num;     atomic number
float              wgt;     atomic weight
int             hybrd;      hybridization type
float          charge;      formal charge
```

CHEMchemunit Object

CHEMchemunit is a second level CHEM"object". A list of one or more CHEMchemunits can be referenced by a CHEMmolecule. The CHEMchemcunit object is used to represent chemical substructures. These substructures could be, for example, amino acid residues, DNA structural units, functional groups, monomer units, or any collection of atoms that need to be represented as an independent unit. The CHEMatoms of a CHEMmolecule can be included in multiple CHEMchemunit instances. However, a CHEMchemunit object can not reference CHEMatoms of other CHEMmolecules. In order for a CHEMchemunit to be properly represented it must be used in conjunction with a valid CHEMint_list lists and CHEMatom linked lists. The CHEMchemunit object is represented by an internal offset list of CHEMatom objects.

CHEMchemunit is composed of the following elements:

```
char                *name;    substructure unit name
int           unit_number;    substructure unit number
CHEMint_list        *atom;    internal atom offsets
                              (0 to (natom-1))
int                parent;    internal offset to
                              atom or molecule.
```

CHEMint_list Object

CHEMint_list is a third level CHEM"object". In order to represent a valid substructure a CHEMint_list must be referenced by a CHEMchemunit. The CHEMint_list object is used to represent an internal offset list for the CHEMchemunit object. This list of offsets is composed of integers greater than or equal to 0 and less than the number of atoms of the parent CHEMmolecule-1.

CHEMint_list is composed of the following element:

```
int                   off;    offset
```

CHEMquantum Object

CHEMquantum is a second level CHEM"object". A list of one or more CHEMquantum objects can be referenced by a CHEMmolecule. The CHEMquantum object is used to represent the quantum information for the parent CHEMmolecule. A valid CHEMatom object list can be used to augment the

CHEMquantum object. Use of the CHEMcandb and CHEMchemunit objects is optional.

The CHEMquantum object is designed to contain information necessary to perform Quantum Mechanical calculations on the parent CHEMmolecule. The CHEMquantum object contains a description of the wavefunction by storing information about the basis set and molecular orbital coefficients. The basis set is assumed to contain gaussian-type functions. For a given basis set, multiple sets of molecular orbital coefficients can be stored through the use of the CHEMq_page object. A CHEMmolecule can have multiple CHEMquantum objects, thus allowing for the description of many different types of calculations for the same molecule. The ancillary CHEMquantum structures are somewhat flexible allowing for variations in these representations; they are adaptable for research and application development. The User_Data node can be extended to address application needs. However, it is up to the application to manage and validate the User_Data representation.

CHEMquantum is composed of the following elements:

```
char            *name;   name of CHEM quantum object
int            nbasis;   number of basis functions
int               ich;   overall charge of molecule
int               mul;   multiplicity
int                ne;   number of electrons
int                na;   number of alpha spin electrons
int                nb;   number of beta spin electrons
int           scftype;   type of SCF
                         (self-consistent field)
                         calculation (RHF for example).
int          corrtype;   type of higher-order calculation
                         (MP2 for example)
int             npage;   number of CHEMq_pages
int            nshell;   number of CHEMshells
int            ngauss;   total number of CHEMgauss
CHEMshell      *shell;   pointer to CHEMshell
                         object list
CHEMq_page        *pl;   pointer to CHEMq_page object list
                         dimensioned nbasis*mbasis;
                         can hold molecular orbital
                         coefficients or other data
char*       User_Data;   for User Data
```

CHEMq_page Object

CHEMq_page is a third level CHEM"object". A list of one or more CHEMq_pages can be referenced by a CHEMquantum. The CHEMq_page object can be used to represent whatever quantities the application requires. As an example, one CHEMq_page can contain the molecular orbital coefficients and the next the density matrix. The CHEMq_page is essentially a matrix for which there are element, row, and column operators. Each CHEMq_page is named. That name is the means to identify its contents. Each coefficient ma-

trix is dimensioned *mbasis* x *nbasis*. The application is responsibile for proper interpretation of the meaning of the matrix via its name.

CHEMq_page is composed of the following elements:

```
char*          name;      name of this page
int          mbasis;      mbasis row
int          nbasis;      nbasis col
double        *page;      dimensioned [mbasis*nbasis]
```

CHEMshell Object

CHEMshell is a third level CHEM"object". A list of one or more CHEMshells must be referenced by a CHEMquantum. The CHEMshell object contains information about a particular gaussian-type basis function. The basis functions are in turn assumed to consist of linear combinations of primitive gaussians. A list of CHEMshell objects would describe the complete basis set used in the calculation. The application is responsible for the management and coherency of the data for the CHEMshell. The CHEMshell object represents a basis set for a given quantum representation. The CHEMshell object is instanced by CHEMquantum object.

CHEMshell is composed of the following elements:

```
char        *name;   shell name
int        kstart;   starting location of the shell within
                     the complete list of primitive gaussians
int         katom;   number of the atom on which the shell resides
int         ktype;   integer description of the type of shell: S, P,
                     D, SP, F... This value is application specific
int           kng;   number of primitive gaussians for this shell
int          kloc;   In the list of basis functions for this molecule,
                     the first function which this shell describes
int          kmin;   starting atomic orbital type
int          kmax;   ending atomic orbital type
CHEMgauss  *gauss;   pointer the CHEMgauss object list.
```

CHEMgauss Object

CHEMgauss is a fourth level CHEM"object". A list of one or more CHEMgauss must be referenced by a CHEMshell. The CHEMgauss object represents the linear combination of primitive gaussian-type functions, centered about the location designated by *coord*, which comprise a particular basis function.

CHEMgauss is composed of the following elements:

```
double        expo;    gaussian exponent
CHEMcoef     *coef;    contraction coefficient
double    coord[3];    cartesian coords of each gaussian
```

CHEMcoef Object

CHEMcoef is a fifth level CHEM"object". A list of one or more CHEMcoef must be referenced by a CHEMgauss object. The CHEMcoef object represents the contraction coefficients for a CHEMcoef function.

CHEMcoef is composed of the following element:

```
double        val;   value of the contraction coefficient
```

CHAPTER 3

CDK NOMENCLATURE

Introduction

The Molecule Data Type provides a significant number of functions to address the needs of application. In order to simplify the use of the CDK, all data, function names, and arguments follow interrelated naming conventions.

Data Type Naming Conventions.

All Molecule Data Type objects are prefixed by "CHEM". As defined in the "Data Types" chapter, CHEM objects include the following:

- CHEMmolecule—Is the root of the Molecule Data Type.
- CHEMatom—Provides the classical representation of an atom,
- CHEMcandb—Provides the connectivity and bond information for an atom.
- CHEMchemunit—Provides the substructures for a given CHEMmolecule.
- CHEMint_list—Supports the description of the substructures for a given CHEMmolecule.
- CHEMquantum—Provides the Quantum Mechanical representation of a given CHEMmolecule.
- CHEMq_page—Supports assorted matrix data for a given CHEMquantum.
- CHEMshell—Supports assorted shell types for a given CHEMquantum.
- CHEMgauss—Defines a primitive function for a given CHEMshell.
- CHEMcoef—Supports the definition of a given CHEMgauss.

Function Naming Conventions.

Each of these objects have two associated classes of function that perform access to, and list manipulation of, the data type in question. Since each CHEM"object" has functions of these two classes and the programmer

knows the contents of the CHEMobject, they can derive each function name given the following formula

- Each "object" is the data type in question
- Each "operator" is the class of function
- Each "element" is the component of the given CHEM"object"

Accessor functions are named this way:

```
CHEM"object"_"operator"_"element"( "args" );
```

Or, when the operator is alloc, free, set, or get:

```
CHEM"object_"operator"("args");
```

List manipulation functions are named this way:

```
CHEM"object"_"operator"( "args" );
```

By mapping out the permutations for the CHEMcandb object, the following chart of possible accessor functions is obtained:

```
          +-   -+ +-    -+
          | get | | conn |
CHEMcandb_|     |_|      |(...);
          | set | | bond |
          +-   -+ +-    -+
```

By mapping out the permutations for the CHEMcandb object, the following chart of possible list manipulation functions is obtained:

```
          +-        -+
          | add      |
CHEMcandb_| del      |(...);
          | insert   |
          | replace  |
          | of_index |
          | get_next |
          | number   |
          | print    |
          +-        -+
```

SINGLE and MULTI

One can link a series of CHEMmolecules together, as shown in Figure 2-1. Various *libchem* molecule and utility functions have a "mode" flag that can be set to either **SINGLE** or **MULTI**. If the function is called with the mode set to **SINGLE** (for example, **CHEMmolecule_extents()**), then the function will apply only to the current molecule. If the mode is set to **MULTI**, then the function will traverse the entire list of molecules and return values for each.

Fortran and C Examples

There is library support for both Fortran and C programmers. Attention has been given to both interfaces to ensure consistent conventions. The following excerpts of code illustrate the relationship between these languages.

C code examples:

```
CHEMcandb *root;
CHEMcandb *cnb1;
int conn;
int bond;
int error;

/* assign the atom's connectivity value*/
conn = 1217;

/* create an instance of CHEMcandb */
cnb1 = CHEMcandb_alloc();

/*set CHEMcandb object connectivity*/
error = CHEMcandb_set_conn( cnb1, conn );

/*get CHEMcandb object connectivity*/
error = CHEMcandb_get_conn( cnb1, &conn );

/* add cnb1 to the list designated by root */
error = CHEMcandb_add( &root, cnb1 );

/*return the next CHEMcandb object */
 cnb1 = CHEMcandb_get_next( &root );

/*print the cnb1 object */
error = CHEMcandb_print( stderr,  &cnb1 );
```

Fortran code examples:

```
      integer root
      integer cnb1
      integer conn
      integer bond
      integer error

C -- assign the atom's connectivity value
      conn = 1217

C -- allocate an instance of CHEMcandb
      cnb1 = CHEMcandb_alloc()

C -- set CHEMcandb object connectivity
      error = chemcandb_set_conn( cnb1, conn )

C -- get CHEMcandb object connectivity
      error = chemcandb_get_conn( cnb1, conn )
```

```
C -- add cnb1 to the list designated by root
     error = chemcandb_add( root, cnb1 )

C -- return the next CHEMcandb object
     cnb1 = chemcandb_get_next( root )

C -- print the cnb1 object
     error = chemcandb_print( chemstdio(),  cnb1 )
```

Argument Passing Conventions

The argument passing is uniform across all functions of a given language. The C and Fortran argument conventions follow.

For C (*libchem)*:

- The list manipulation functions expect that the root of a CHEM"object" list is the address of that CHEM"object" pointer, (CHEM"object"** or &CHEM"object"*). All other arguments to list manipulations functions expect CHEM"object" pointers, (CHEM"object"*), or value arguments.
- All accessor set functions expect the CHEM"object" pointers and value arguments.
- All accessor get functions expect the CHEM"object" pointers and pointer arguments for the data to be returned.
- All CHEM"object" allocation functions return CHEM"object" pointers.
- All CHEM"object"_free functions expect CHEM"object" pointer arguments.

For Fortran (*libchem_f*):

- The list manipulation functions expect that CHEM"object"s pointers and roots are integers. The Fortran library, *libchem_f*, insures that the proper arguments are passed to and returned from the underlying C library libchem_c.
- All floats are real.
- All doubles are real*8.
- All strings should be defined as character arrays. The CDK underlying dynamic string management will truncate or pad the character array appropriately. Fortran users should use sufficiently large character arrays.

General Utility Functions

The third class of CDK function, general utility, provides translation and management of CDK data and memory. These functions must be reviewed individually in the "CDK Library Reference" chapter's "General Utlity Interface Functions" section.

CHAPTER 4

CDK PROGRAMMER NOTES

Programmer Notes Introduction

The following notes and observations have been compiled during the use and development of the CDK. This information can help you to avoid bugs and misconceptions about the CDK.

General

- In order to ensure the forward compatibilty of any application, one must use the functions provided and not refer to any element of, or create a CHEM"object" directly.
- The **CHEMmolecule_copy** function copies one molecule structure to another: there are no other ancillary CHEM"object"_copy functions.
- CHEM"object" set functions, which set the values of elements in a list, check to see if the list has been initialized. If the list has not been initialized, the set function will return an error and set the element values to 0.
- There are no facilities to propagate changes or send messages from one CHEM type to another. For example, if one atom of the CHEMatom list is deleted, the number of atoms, *natom*, reflected in CHEMmolecule is not changed.
- Use of the list manipulation facilities will fail if a list is added, inserted or replaced by any subsegment of itself. Effectively, this will create a list that points to itself, resulting in non-terminating lists. Circular lists are NOT supported.
- In all cases, the programmer must use the provided CHEM"object" allocation and free functions. Use of any flavor of *malloc* and *free* directly may result in hard-to-resolve memory corruptions.
- It is possible that modules from different origins will expect different User Data definitions. It is the responsibility of the application to ensure that the User Data representation received is correct.

Fortran Specific

- The Fortran string interfaces convert all dynamic strings into character arrays. The arrays may be declared any size in the Fortran subroutine.
- All indices that are passed by Fortran users to *libchem* must be 0 based.
- Fortran modules that expect input molecules must call the function *molecule* = **chemgen_util_input_molecule(***mol_input***)**, where *mol_input* is passed in to the compute function and *molecule* is an integer that represents a CHEMmolecule object.
- Programmers should include *cheminc/CHEMlong.inc* on systems that do not limit the length of symbol names, and *cheminc/CHEMshrt.inc* on systems that limit symbol names to eight characters or less.

C Specific

- The internal string management facilities, accessible by C, will fail if they are passed any char arrays or if the value of the initial char* is not initialized to NULL.
- C programmers must include the line **#define CHEM_APPL** before including *CHEMmol.h.*
- C programmers wishing to #define the constants **ATOM** and **MOLECULE** will find they are defined already in the include file */usr/avs/include/chemistry/CHEMgen_lm.h.*

CHAPTER 5

CDK LIBRARY REFERENCE

Introduction

This chapter describes all calls available in the CDK.

CDK Library Notes

All CHEM"object"s—molecule, atom, candb, chemunit, int_list, quantum, shell, q_page, gauss, and coef—use the same facilities for manipulating lists. A list of objects is defined to be a **linked list** of homogeneous objects with one or more members. The start of this list is called the **root**, the individual objects of the list are **members**, and the last object is the terminating object. Each CHEM"object" has a set of operations to create, manipulate and delete objects of its class. For each CHEM"object" the following functions are defined: add, free, insert, replace, of_index, num, get_next, and print. These functions focus exclusively on manipulating and querying CHEM"object" lists.

As described in the "Nomenclature" chapter, each object manipulation function may be constructed from its object name and the class of operation requested.

Each of these functions is valid only if they operate on CHEM"objects" of their own type.

Since the functions listed in the "CDK List Manipulation Functions" sections are defined in terms of CHEM"object"s, it is important to remember that the term "object" should be replaced with the appropriate CHEM type. In order to map the function required into its real name, merely replace "object" with any of the following types: molecule, atom, candb, chemunit, int_list, quantum, shell, q_page, gauss, or coef.

It is important to note that if a list, or any subset of itself, is added to, inserted in or replaced by a list of its members, then the user may have defined a circular list. This practice is certain to cause problems. The user should be careful NOT to build curcular lists.

CDK Error Returns

All functions return standard CHEMerror codes unless otherwise stated. It is the responsibility of the application to check for these codes and act upon them appropriately. The codes are listed below:

- **CHEM_NOERR**—no error.
- **CHEM_ERRAUD**—CHEMatom internal user data error.
- **CHEM_ERRNIL**—CHEM"object" not in list.
- **CHEM_ERRNP**—CHEM"object" is a NULL.
- **CHEM_ERR**—General accessor error.

CDK List Manipulation Functions

The list manipulation functions for all CHEM"object"s are abstractly described below. Each of the list manipulation functions listed is valid for each CHEM"object". In order to directly call a particular CHEM"object" list manipulation function it is necessary to substitute the appropriate data type name for the "object" phrase. For example, to add CHEMmolecule object to a list, one must translate CHEM"object"_add(root, obj); into **CHEMmolecule_add(** *arg1, arg2***)**;

CHEMobject_add

```
C                                       Fortran
int                                     integer
CHEM"object"_add(root, obj)             chem"object"_add(root, obj)

CHEM"object"  **root;                   integer   root
CHEM"object"  *obj;                     integer   obj
```

CHEM"object"_add will add the CHEM"object" obj, to the end of the CHEM"object" list designated by root. If the list designated by root has not been defined, (is NULL or 0), then the root of the list is initialized by the list obj.

CHEMobject_del

```
C                                       Fortran
int                                     integer
CHEM"object"_del(root, obj)             chem"object"_del(root, obj)

CHEM"object"  **root;                   integer    root
CHEM"object"  *obj;                     integer    obj
```

CHEM"object"_del will delete and free a single member of a list, obj, from the CHEM"object" list designated by root. In addition CHEM"object"_del will resolve references to obj within the the list designated by root. Should obj be NULL or 0 then CHEM"object"_del will delete the terminating member of the list.

CHEMobject_insert

```
C                                       Fortran
int                                     integer
CHEM"object"_insert(root, targ, obj)    chem"object"_insert(root,targ,obj)

CHEM"object"  **root;                   integer    root
CHEM"object"  *targ;                    integer    targ
CHEM"object"  *obj;                     integer    obj
```

Given the root of a valid CHEM"object" list, CHEM"object"_insert will search for the member targ after which it will insert the list obj. If any of the arguments are NULL or 0 the insert operation is aborted.

CHEMobject_replace

```
C                                       Fortran
int                                     integer
CHEM"object"_replace(root, targ, obj)   chem"object"_replace(root,targ,obj)

CHEM"object"  **root;                   integer    root
CHEM"object"  *targ;                    integer    targ
CHEM"object"  *obj;                     integer    obj
```

Given the root of a valid CHEM"object" list, CHEM"object"_replace will search for the member targ and it replace it with the CHEM"object" list, obj. If any of the arguments are NULL or 0 the replace operation is aborted.

CHEMobject_num

```
C                                       Fortran
int                                     integer
CHEM"object"_num(root, targ)            chem"object"_num(root, targ)

CHEM"object"  **root;                   integer    root
CHEM"object"  *targ;                    integer    targ
CHEM"object"  *obj;                     integer    obj
```

Given the root of a valid CHEM"object" list and a member of that list targ, CHEM"object"_num will return the integer offset of targ in the list. If the root

is NULL or 0 then CHEM"object"_num will return CHEM_ERR. If the target is not present in the list the **CHEM_NIL** is returned. If targ is defined to be NULL or 0, then CHEM"object"_num will return the total number of members in the list.

CHEMobject_of_index

```
C                                       Fortran
CHEM"object"*                           integer
CHEM"object"_of_index(root, num)        chem"object"_of_index(root, num)

CHEM"object"  **root;                   integer    root
int           num;                      integer    num
```

Given the root of a valid CHEM"object" and an integer offset, CHEM"object"_of_index will return the CHEM"object" of the nth CHEM"object" in the list. If the argument num is less than 0 or greater than the total number of molecules in the list, then CHEM"object"_of_index will return NULL.

CHEMobject_get_next

```
C                                       Fortran
CHEM"object"*                           integer
CHEM"object"_get_next(root)             chem"object"_get_next(root)

CHEM"object"  **root;                   integer    root
```

Given the root of a valid CHEM"object" list, CHEM"object"_get_next will return the next member in the list.

CHEMobject_print

```
C                                       Fortran
int                                     integer
CHEM"object"_print(fp, root)            CHEM"object"_print(fp, root)

FILE          *fp;                      integer    fp
CHEM"object"  **root;                   integer    root
```

CHEM"object"_print will traverse and print all data and descendant CHEM"object"s to the file designated by fp. Fortran use of this function requires that fp be created by either the functions **chemgen_util_stderr**, **chemgen_util_stdout**, or **chemgen_util_open_file(** *name*, *mode* **).** These functions are defined in the "General Utility Interface Section" of this chapter. C programmers need not use these functions; they may rely directly on *stdout*, *stderr* or other FILE utilities.

The ASCII output produced by these functions is not recommended as a format for long term storage; it is intended as a means to review data. Subsequently these functions will not print user data, and there is no associated parser to read print data written to disk.

Molecule Interface Functions

The CHEMmolecule object is the root of any AVS Molecule Data Type structure. The user data node may be extended to fill application needs; however it is up to the application to ensure that the information is valid and that mechanisms are provided for its access and manipulation. It is composed of the following components:

```
char                *name;   molecule name
int                 units;   units: ANGSTROMS or BOHRS
int                 natom;   number of CHEMatom objects
int               nc_unit;   number of CHEMchemunit objects
int                nquant;   number of CHEMquantum objects
CHEMatom            *atom;   pointer to CHEMatom object list
CHEMchemunit      *c_unit;   pointer to CHEMchem_unit object list
CHEMquantum        *quant;   pointer to CHEMquantum object list
char           *User_Data;   User Data node
```

CHEMmolecule_alloc

```
C                                   Fortran
CHEMmolecule*                       integer
CHEMmolecule_alloc()                chemmolecule_alloc()
```

The **CHEMmolecule_alloc** function returns a CHEMmolecule object to the caller. All data is initialized to 0, with no ancillary objects. These Fortran and C functions will not return a CHEM error code; rather they will return 0 or NULL respectively.

CHEMmolecule_free

```
C                               Fortran
void                            integer
CHEMmolecule_free(mol)                 chemmolecule_free(mol)

CHEMmolecule     *mol;                 integer    mol
```

Given a valid CHEMmolecule object, mol, **CHEMmolecule_free** will traverse and free itself and all ancillary objects. It will not traverse and delete a list of CHEMmolecule objects. To delete and free the entire list, use **CHEMmolecule_del_col**.

CHEMmolecule_get

```
C                                       Fortran
int                                     integer
CHEMmolecule_get(mol, name,             chemmolecule_get(mol, name,
     units, natom, nc_unit,                  units, natom, nc_unit,
     nquant, atom_l, chem_l,                  nquant, atom_l, chem_l,
     quant_l, udat)                         quant_l, udat)

CHEMmolecule  *mol;                     integer       mol
char          **name;                   character*256 name
int           *units;                   integer           units
int           *natom;                   integer           natom
int           *nc_unit;                 integer           nc_unit
int           *nquant;                  integer           nquant
CHEMatom      **atom_l;                 integer       atom_l
CHEMchemunit  **chem_l;                 integer       chem_l
CHEMquantum   **quant_l;                integer       quant_l
char          **udat;                   integer       udat;
```

Given a valid CHEMmolecule object, mol, **CHEMmolecule_get** will return all the components of mol. Any data that has not been previously set will be returned as 0, and any uninitialized ancillary object lists will be terminated. Refer to the *AVS Developer's Guide* for more information regarding the use of user data.

CHEMmolecule_set

```
C                                       Fortran
int                                     integer
CHEMmolecule_set(mol, name,             chemmolecule_set(mol,  name,
       units, natom, nunit,                    units, natom, nunit,
    nquant, atom_l, unit_l,                 nquant, atom_l, unit_l,
             quant_l, udat)                          quant_l, udat)

CHEMmolecule  *mol;                     integer       mol
char          *name;                    character*256 name
int           units;                    integer           units
int           natom;                    integer           natom
int           nunit;                    integer           nunit
int           nquant;                   integer           nquant
CHEMatom      *atom_l;                  integer       atom_l
CHEMchemunit  *unit_l;                  integer       unit_l
CHEMquantum   *quant_l;                 integer       quant_l
char          *udat;                    integer       udat
```

Given a valid CHEMmolecule object mol, **CHEMmolecule_set** will set all the components of mol. Care should be taken to ensure the user data component of the CHEMmolecule object is created using the proper utilities. Refer to the *AVS Developer's Guide* for more information regarding the use of user data.

CHEMmolecule_get_name

```
C                                       Fortran
int                                     integer
CHEMmolecule_get_name(mol, name)        chemmolecule_get_name(mol, name)

CHEMmolecule  *mol;                     integer       mol
char          **name;                   character*256 name
```

Given a valid CHEMmolecule object, mol, **CHEMmolecule_get_name** will return the name component of mol.

CHEMmolecule_set_name

```
C                                       Fortran
int                                     integer
CHEMmolecule_set_name(mol, name)        chemmolecule_set_name(mol, name)

CHEMmolecule  *mol;                     integer       mol
char          *name;                    character*256 name
```

Given a valid CHEMmolecule object, mol, **CHEMmolecule_set_name** will set the name component of mol.

CHEMmolecule_get_user_data

```
C                                       Fortran
int                                     integer
CHEMmolecule_get_user_data(mol, udat)   chemmolecule_get_user_data(mol,
udat)

CHEMmolecule    *mol;                   integer       mol
char            **udat;                 integer       udat
```

Given a valid CHEMmolecule object mol, **CHEMmolecule_get_user_data** will return the user data component of mol. In order for this information to be valid the user must have registered the AVS user data type. Failure to adhere to the User Data model, outlined in the *AVS Developer's Guide* manual will result in the defined type not being communicated in an AVS network.

CHEMmolecule_set_user_data

```
C                                       Fortran
int                                     integer
CHEMmolecule_set_user_data(mol, udat)   chemmolecule_set_user_data(mol,udat)
```

```
CHEMmolecule  *mol;                    integer       mol
char          *udat;                   integer       udat
```

Given a valid CHEMmolecule object mol, **CHEMmolecule_set_user_data** will return the user data component of mol. In order for this information to be valid the user must have registered the AVS user data type. Failure to adhere to the User Data model, outlined in the *AVS Developer's Guide* manual will result in the defined type not being communicated in an AVS network.

CHEMmolecule_get_units

```
C                                      Fortran
int                                    integer
CHEMmolecule_get_units(mol, units)     chemmolecule_get_units(mol, units)

CHEMmolecule  *mol;                    integer       mol
int           *units;                  integer       units
```

Given a valid CHEMmolecule object mol, **CHEMmolecule_get_units** will return the units component of mol. By definition this may be either **ANGSTROMS**, the default, or **BOHRS**.

CHEMmolecule_set_units

```
C                                      Fortran
int                                    integer
CHEMmolecule_set_units(mol, units)     chemmolecule_set_units(mol, units)

CHEMmolecule  *mol;                    integer       mol
int           units;                   integer       units
```

Given a valid CHEMmolecule object mol, **CHEMmolecule_set_units** will set the units component of mol to either **ANGSTROMS** or **BOHRS**. This parameter is constrained to either **ANGSTROMS** or **BOHRS** and defaults to **ANGSTROMS** when given invalid arguments.

CHEMmolecule_get_natom

```
C                                      Fortran
int                                    integer
CHEMmolecule_get_natom(mol, natom)     chemmolecule_get_natom(mol, natom)

CHEMmolecule  *mol;                    integer       mol
int           *natom;                  integer       natom
```

Given a valid CHEMmolecule object mol, **CHEMmolecule_get_natom** will return the the number of atoms in the CHEMatom object list previously set by the user. The number of atoms is constrained to be a non-negative integer. This function does not traverse the atom object list and return the current count; it is dependent on a prior call to **CHEMmolecule_set_natom**.

CHEMmolecule_set_natom

```
C                                       Fortran
int                                     integer
CHEMmolecule_set_natom(mol, natom)      chemmolecule_set_natom(mol, natom)

CHEMmolecule  *mol;                     integer       mol
int           natom;                    integer       natom
```

Given a valid CHEMmolecule object mol, **CHEMmolecule_set_natom** will set the the natom component of mol. This integer value is constrained to be a non-negative integer.

CHEMmolecule_get_nunit

```
C                                       Fortran
int                                     integer
CHEMmolecule_get_nunit(mol, nunit)      chemmolecule_get_nunit(mol, nunit)

CHEMmolecule  *mol;                     integer       mol
int            *nunit;                   integer       nunit
```

Given a valid CHEMmolecule object mol, **CHEMmolecule_get_nunit** will return the number of chemunits, nunit, component previously set by the user. The number of chemunits is constrained to be a non-negative integer. This function does not traverse the CHEMchemunit object list and return the current count. It is dependent on a prior call to **CHEMmolecule_set_nunit**.

CHEMmolecule_set_nunit

```
C                                       Fortran
int                                     integer
CHEMmolecule_set_nunit(mol, nunit)      chemmolecule_set_nunit(mol, nunit)

CHEMmolecule  *mol;                     integer       mol
int           nunit;                    integer       nunit
```

Given a valid CHEMmolecule object mol, **CHEMmolecule_set_nunit** will set the number of chemunits component of mol. This integer value is constrained to be a non-negative integer.

CHEMmolecule_get_nquant

```
C                                       Fortran
int                                     integer
CHEMmolecule_get_nquant(mol, nquant)    chemmolecule_get_nquant(mol,
nquant)

CHEMmolecule  *mol;                     integer       mol
int           *nquant;                  integer       nquant
```

Given a valid CHEMmolecule object mol, **CHEMmolecule_get_nquant** will return the the number of quanta component previously set by the user. The number of quanta is constrained to be a non-negative integer. This function does not traverse the CHEMquantum object list and return the current count; it is dependent on a call to **CHEMmolecule_set_nquant**.

CHEMmolecule_set_nquant

```
C                                       Fortran
int                                     integer
CHEMmolecule_set_nquant(mol, nquant)    chemmolecule_set_nquant(mol,
nquant)

CHEMmolecule  *mol;                     integer       mol
int           nquant;                   integer       nquant
```

Given a valid CHEMmolecule object mol, **CHEMmolecule_set_nquant** will set the number of quanta. This integer value is constrained to be a non-negative integer.

CHEMmolecule_get_atom

```
C                                       Fortran
int                                     integer
CHEMmolecule_get_atom(mol, atom)        chemmolecule_get_atom(mol, atom)

CHEMmolecule* mol;                      integer      mol
CHEMatom** atom;                        integer      atom
```

Given a valid CHEMmolecule object mol, **CHEMmolecule_get_atom** will return mol's CHEMatom object list.

CHEMmolecule_set_atom

```
C                                       Fortran
int                                     integer
CHEMmolecule_set_atom(mol, atom)        chemmolecule_set_atom(mol, atom)

CHEMmolecule*  mol;                     integer       mol
CHEMatom*      atom;                    integer       atom
```

Given a valid CHEMmolecule object mol, **CHEMmolecule_set_atom** will set mol's CHEMatom object list to that described by the argument atom. The CHEMatom object list should be composed of atoms objects created by calls to **CHEMatom_alloc**.

CHEMmolecule_get_chem

```
C                                       Fortran
int                                     integer
CHEMmolecule_get_chem(mol, chem)        chemmolecule_get_chem(mol, chem)

CHEMmolecule*  mol;                     integer      mol
CHEMchemunit** chem;                    integer      chem
```

Given a valid CHEMmolecule object mol, **CHEMmolecule_get_chem** will return mol's CHEMchemunit object list.

CHEMmolecule_set_chem

```
C                                       Fortran
int                                     integer
CHEMmolecule_set_chem(mol, chem)        chemmolecule_set_chem(mol, chem)

CHEMmolecule*  mol;                     integer       mol
CHEMchemunit*  chem;                    integer       chem
```

Given a valid CHEMmolecule object mol, **CHEMmolecule_set_chem** will set mol's CHEMchemunit object list to that described by the argument chem. The CHEMchemunit object list should be composed of chemunit objects created by calls to **CHEMchemunit_alloc**.

CHEMmolecule_get_quant

```
C                                       Fortran
int                                     integer
CHEMmolecule_get_quant(mol, quant)      chemmolecule_get_quant(mol, quant)
```

```
CHEMmolecule*  mol;                    integer      mol
CHEMquantum**  quant;                  integer      quant
```

Given a valid CHEMmolecule object mol, **CHEMmolecule_get_chem** will return mol's CHEMquantum object list.

CHEMmolecule_set_quant

```
C                                      Fortran
int                                    integer
CHEMmolecule_set_quant(mol, quant)     chemmolecule_set_quant(mol, quant)

CHEMmolecule*  mol;                    integer       mol
CHEMquantum*   quant;                  integer       quant
```

Given a valid CHEMmolecule object mol, **CHEMmolecule_set_quant** will set mol's CHEMquantum object list to that described by the argument quant. The CHEMquantum object list should be composed of quantum objects created by calls to **CHEMquantum_alloc**.

CHEMmolecule_copy

```
C                                      Fortran
int                                    integer
CHEMmolecule_copy(m_in, mode, m_out)   chemmolecule_copy(m_in, mode, m_out)

CHEMmolecule  **m_in;                  integer       m_in
int           mode;                    integer       mode
CHEMmolecule  **m_out;                 integer       m_out
```

Given the root of a valid CHEMmolecule list, m_in, a mode flag either (**MULTI/SINGLE**) (1, 0) and a initialized CHEMmolecule object m_out, **CHEMmolecule_copy** will traverse and create a copy of the CHEMmolecule object mol_in into mol_out. If the mode is set to be **MULTI** then the function will traverse and copy the entire molecule list. If the mode is set to be **SINGLE** only a copy of the member m_in is performed. There are no ancillary CHEM"object" copy functions: only CHEMmolecule objects may be copied in their entirety. The CHEMmolecule copy, m_out, may be dissected and portions of it preserved as a work around for lack of the ancillary CHEM"object" copy functions.

CHEMmolecule_extents

```
C                                      Fortran
int                                    integer
```

```
CHEMmolecule_extents(mol, mode, ext)    chemmolecule_extents(mol, mode, ext)

CHEMmolecule  *mol;                     integer       mol
int           mode;                     integer       mode
float         ext[6];                   real          ext[6]
```

Given a valid CHEMmolecule object mol, a mode flag either **MULTI** or **SINGLE**, **CHEMmolecule_extents** will return the minimum bounding extents describing mol. The order of extents in the float array is: xmin, ymin, zmin, xmax, ymax, and zmax. If the mode is set to be **MULTI** then the function will traverse and compute the extents of the entire CHEMmolecule list. If the mode is set to be **SINGLE** only the extents of member mol are computed. **CHEMmolecule_extents** takes into account the radius of each atom.

CHEMmolecule_cubic_extents

```
C                                       Fortran
int                                     integer
CHEMmolecule_cubic_extents(ex, mn, mx)  chemmolecule_cubic_extents(ex, mn, mx)

float        ex[6];                     real       ex[6]
float        *mn;                       real       mn
float        *mx;                       real       mx
```

Given a float array, ex, containing the extents of a CHEMmolecule list, **CHEMmolecule_cubic_extents** will return the minimum, mn, and maximum, mx, values of the extent array. These values may be used to construct a minimum bounding cube of the CHEMmolecule list.

CHEMmolecule_del_col

```
C                                       Fortran
int                                     integer
CHEMmolecule_del_col(mol)               chemmolecule_del_col(mol)

CHEMmolecule **mol;                     integer mol
```

Given the root of a CHEMmolecule, mol, **CHEMmolecule_del_col** will traverse the list of molecules designated by mol and delete and free the entire list. **CHEMmolecule_del_col** will traverse and free all ancillary CHEMobjects as well. (To delete and free individual molecules, see **CHEMmolecule_free**.)

CHEMatom Interface Functions

The CHEMatom object is a classical atom representation. It is instanced by the CHEMmolecule object. It may be used in conjunction with a CHEMquantum object if desired, and it is necessary in defining valid CHEMchemunit objects. The CHEMcandb object list is necessary in order to define connectivity and bond information; some application's may require this information while others may not. The components of the CHEMatom object follow:

```
int           index_number;    number as assigned by the application.
char                 *name;    name
int                  color;    color
double            x, y, z;    position
float              radius;    radius
CHEMcandb             *cnb;    pointer to CHEMcandb object list
char            *User_Data;    User Data node
```

The internally defined CHEMudata structure addresses general needs of an extended atom structure. Should the user require a variation on what is presented here then they should directly create their own User Data structure and provide for its access and management. The components of the internally defined user data structure follow:

```
int parent;     parent molecule
int a_num;      atomic number
float wgt;      atomic weight
int hybrd;      hybridization type
double charge;  formal charge
```

The instance of User Data allows the application to extend the CHEMatom object definition; however it is up to the application to check and ensure that the information is valid. The default state is no user data. It is the application's responsibility to define manage and access the externally defined user data nodes. The internal definition is defined and managed by the CDK. The internal User Data component is accessed by a set of functions that extend the definition of the CHEMatom object. Should the application determine that the internal user data representation will suffice, then a function, **CHEMatom_init_user_data**, may be called to initialize and make available the CHEMatom functions that access the internal user_data structure. Those CHEMatom functions that rely on the internal user_data representation are clearly marked.

CHEMatom_alloc

```
C                                   Fortran
CHEMatom*                           integer
CHEMatom_alloc()                    chematom_alloc()
```

The **CHEMatom_alloc** function returns a CHEMatom object to the caller. All data is initialized to 0, with no ancillary objects. These Fortran and C func-

tions will not return a CHEM error code; rather they will return 0 or NULL respectively.

CHEMatom_free

```
C                                       Fortran
void                                    integer
CHEMatom_free(atom)                     CHEMatom_free(atom)

CHEMatom     *atom;                     integer    atom
```

Given a valid CHEMatom object atom, **CHEMatom_free** will traverse the list designated by atom and free all ancillary CHEMcandb objects.

CHEMatom_get

```
C                                       Fortran
int                                     integer
CHEMatom_get(atom, index_number,        chematom_get(atom, index_number,
   name, color, x, y, z,                    name, color, x, y, z,
   radius, cb, ud)                          radius, cb, ud)

CHEMatom     *atom;                     integer        atom
int          *index_number;             integer        index_number
int          *color;                    integer        color
char         **name;                    character*256  name
double       *x, *y, *z;                real*8         x, y, z
float        *radius;                   integer        radius
CHEMcandb    **cb;                      integer        cb
char         **ud;                      integer        ud
```

Given a valid CHEMatom object, atom, **CHEMatom_get** will return all components of object atom. Any data that has not been previously set will be returned as 0, and any uninitialized ancillary object lists will be terminated.

CHEMatom_set

```
C                                       Fortran
int                                     integer
CHEMatom_set(atom, index_number,        chematom_set(atom, index_number,
   name, color, x, y, z,                    name, color, x, y, z,
   radius, cb, ud)                          radius, cb, ud)

CHEMatom     *atom;                     integer        atom
int          index_number;              integer        index_number
int          color;                     integer        color
char         *name;                     character*256  name
double       x, y, z;                   real*16        x, y, z
```

```
float        radius;                    integer        radius
CHEMcandb    *cb;                       integer        cb
char         *ud;                       integer        ud
```

Given a valid CHEMatom object, atom, **CHEMatom_set** will set all components of object atom. Setting values to NULL or 0 respectively will ensure that ancillary lists are unused.

CHEMatom_get_inumber

```
C                                       Fortran
int                                     integer
CHEMatom_get_inumber(atom, i_num)       chematom_get_inumber(atom, i_num)

CHEMatom     *atom;                     integer       atom;
int          *i_num;                    integer       i_num;
```

Given a valid CHEMatom object atom, **CHEMatom_get_inumber** will return the index number, i_num, component of the atom. The number is for programmer / user reference, and not indicative of any internal list.

CHEMatom_set_inumber

```
C                                       Fortran
int                                     integer
CHEMatom_set_inumber(atom, i_num)       chematom_set_inumber(atom, i_num)

CHEMatom     *atom;                     integer       atom;
int          i_num;                     integer       i_num;
```

Given a valid CHEMatom object atom, **CHEMatom_set_inumber** will set the index number, i_num, of the atom. The number is for programmer / user reference, and not indicative of any internal list; it is, however, constrained to be greater or equal to 0.

CHEMatom_get_name

```
C                                       Fortran
int                                     integer
CHEMatom_get_name(atom, name)           chematom_get_name(atom, name)

CHEMatom  *atom;                        int            atom;
char      **name;                       character*256  name;
```

Given a valid CHEMatom object atom, **CHEMatom_get_name** will return the name component of atom.

CHEMatom_set_name

```
C                                       Fortran
int                                     integer
CHEMatom_set_name(atom, name)           chematom_set_name(atom, name)

CHEMatom  *atom;                        int            atom;
char      *name;                        character*256  name;
```

Given a valid CHEMatom object atom, **CHEMatom_set_name** will set the name component of atom.

CHEMatom_get_xyz

```
C                                       Fortran
int                                     integer
CHEMatom_get_xyz(atom, x, y, z)         chematom_get_xyz(atom, x, y, z)

CHEMatom     *atom;                     integer     atom
double       *x, *y, *z;                real*8      x, y, z
```

Given a valid CHEMatom object atom, **CHEMatom_get_xyz** will return the x, y and z coordinate components of atom.

CHEMatom_set_xyz

```
C                                       Fortran
int                                     integer
CHEMatom_set_xyz(atom, x, y, z)         chematom_set_xyz(atom, x, y, z)

CHEMatom     *atom;                     integer     atom
double       x, y, z;                   real*8      x, y, z
```

Given a valid CHEMatom object atom, **CHEMatom_set_xyz** will assign the x, y and z coordinate components of atom.

CHEMatom_get_radius

```
C                                       Fortran
int                                     integer
CHEMatom_get_radius(atom, radius)       chematom_get_radius(atom, radius)
```

```
CHEMatom    *atom;                      integer    atom
float       *radius;                    real       radius
```

Given a valid CHEMatom object atom, **CHEMatom_get_radius** will return the radius component of atom.

CHEMatom_set_radius

```
C                                       Fortran
int                                     integer
CHEMatom_set_radius(atom, radius)       chematom_set_radius(atom, radius)

CHEMatom *atom;                         integer    atom
float    radius;                        real       radius
```

Given a valid CHEMatom object atom, **CHEMatom_set_radius** will assign the radius component of atom. Radius is constrained to be greater than 0.0.

CHEMatom_get_color

```
C                                       Fortran
int                                     integer
CHEMatom_get_color(atom, color)         chematom_get_color(atom, color)

CHEMatom     *atom;                     integer    atom
int          *color;                    integer    color
```

Given a valid CHEMatom object atom, **CHEMatom_get_color** will return the color component of atom. There are conversion functions which convert integer to rgb values between 0.0 and 1.0 and vise versa. They may be used to easily convert from one format to another. Bear in mind that the rgb components of integer colors will vary from machine to machine. It is advisable to avoid explicit assignment of integer colors. For example, on some machines the high order bits 0xFF0000 will define red, while on others they may define blue.

CHEMatom_set_color

```
C                                       Fortran
int                                     integer
CHEMatom_set_color(atom, color)         chematom_set_color(atom, color)

CHEMatom     *atom;                     integer atom
int          color;                     integer color
```

Given a valid CHEMatom object atom, **CHEMatom_get_color** will assign the color component of atom. There are conversion functions which convert rgb values between 0.0 and 1.0 to integer values and vise versa. They may be used to easily convert from one format to another. Bear in mind that the rgb components of integer colors will vary from machine to machine. It is advisable that explicit assignment of integer colors be avoided. For example, on some machines the high order bits 0xFF0000 will define red, while on others they may define blue.

CHEMatom_get_candb

```
C                                       Fortran
int                                     integer
CHEMatom_get_candb(atom, cnb)           CHEMatom_get_candb(atom, cnb)

CHEMatom     *atom;                     integer atom
CHEMcandb    **cnb;                     integer candb
```

Given a valid CHEMatom object atom, **CHEMatom_get_candb** will return the atom's connectivity and bond list component.

CHEMatom_set_candb

```
C                                       Fortran
int                                     integer
CHEMatom_set_candb(atom, cnb)           chematom_set_candb(atom, cnb)

CHEMatom     *atom;                     integer atom
CHEMcandb    *cnb;                      integer cnb
```

Given a valid CHEMatom object atom, **CHEMatom_set_candb** will assign the atom's connectivity and bond list component to be cnb.

CHEMatom_get_user_data

```
C                                       Fortran
int                                     integer
CHEMatom_get_user_data(atom, ud)        chematom_get_user_data(atom, ud)

    CHEMatom    *atom;                  integer atom
    char        **ud;                   integer ud
```

Given a valid CHEMatom object atom, **CHEMatom_get_user_data** returns the user_data component of atom. In order for this information to be valid the user must have registered the AVS user data type. Failure to adhere to the

User Data model, outlined in the *AVS Developer's Guide* manual will result in the defined type not being communicated in an AVS network.

CHEMatom_set_user_data

```
C                                       Fortran
int                                     integer
CHEMatom_set_user_data(atom, ud)        chematom_set_user_data(atom, ud)

CHEMatom     *atom;                     integer atom
CHEMudata    *ud;                       integer ud
```

Given a valid CHEMatom object atom, **CHEMatom_set_user_data** will assign the user data component of atom. Setting ud without following the prescribed procedures for registering User Data will cause CHEM errors.

CHEMatom_init_user_data

```
C                                       Fortran
int                                     integer
CHEMatom_init_user_data()               CHEMatom_init_user_data()
```

CHEMatom_init_user_data registers the internal CHEMatom user data type, CHEMudata, for use within the AVS framework. A call to this function is necessary in order to use internal user_data representation and its associated functions. The chemistry library, *libchem*, initializes an additional set of CHEMatom functions for accessing the internal user data representation. These functions extend the CHEMatom definition interfaces to include all components of the CHEMudata data type; it must be called once at the beginning of a module that wishes to use the internal representation of user data.

CHEMatom_alloc_user_data

```
C                                       Fortran
int                                     integer
CHEMatom_alloc_user_data(atom)          chematom_alloc_user_data(atom)

CHEMatom *atom                          integer atom
```

The **CHEMatom_alloc** function creates and initializes the internal user data component for the CHEMatom, atom. All data is initialized to 0. This function is valid only if **CHEMatom_init_user_data** has been previously called, and it must be called before there is any assignment to the internal user data components.

CHEMatom_get_user_data_data

```
C                                       Fortran
int                                     integer
CHEMatom_get_user_data_data(atom,       chematom_get_user_data_data(atom,
    parent, a_num, wgt,                         parent, a_num, wgt,
    hybrd, charge)                              hybrd, charge)

    CHEMatom     *atom;                 integer  atom
    int          *parent;               integer  parent
    int          *a_num;                integer  a_num
    float        *wgt;                  real     wgt
    int          *hybrd;                int      hybrd
    double       *charge ;              real*8   charge
```

Given a valid CHEMatom object atom, **CHEMatom_get_user_data_data** returns all associated internal user data components for atom. This function is valid only if **CHEMatom_init_user_data** has been previously called.

CHEMatom_set_user_data_data

```
C                                       Fortran
int                                     integer
CHEMatom_set_user_data_data(atom,       chematom_set_user_data_data(atom,
    parent, a_num, wgt,                    parent, a_num, wgt,
    hybrd, charge)                         hybrd, charge)

CHEMatom     *atom;                     integer  atom
int          parent;                    integer  parent
int          a_num;                     integer  a_num
float        wgt;                       real     wgt
int          hybrd;                     int      hybrd
double       charge ;                   real*8   charge
```

Given a valid CHEMatom object atom, **CHEMatom_set_user_data_data** assigns all associated internal user data components for atom. This function is valid only if **CHEMatom_init_user_data** has been previously called.

CHEMatom_get_parent

```
C                                       Fortran
int                                     integer
CHEMatom_get_parent(atom, parent)       chematom_get_parent(atom, parent)

CHEMatom       *atom;                   integer atom
int            *parent;                 integer parent
```

Given a valid CHEMatom object atom, **CHEMatom_get_parent** returns the integer offset to the CHEMmolecule of which atom is a member. Parent is constrained to be an integer greater or equal to 0. This function is valid only if **CHEMatom_init_user_data** has been previously called.

CHEMatom_set_parent

```
C                                       Fortran
int                                     integer
CHEMatom_set_parent(atom, parent)       chematom_set_parent(atom, parent)

   CHEMatom       *atom;                integer atom
   int            parent;               integer parent
```

Given a valid CHEMatom object atom, **CHEMatom_set_parent**, assigns the integer offset to the CHEMmolecule of which atom is a member. Parent is constrained to be integer greater than or equal 0. This function is valid only if **CHEMatom_init_user_data** has been previously called.

CHEMatom_get_a_num

```
C                                       Fortran
int                                     integer
CHEMatom_get_a_num(atom, a_num)         chematom_get_a_num(atom, a_num)

CHEMatom      *atom;                    integer atom
int           *a_num;                   integer a_num
```

Given a valid CHEMatom object atom, **CHEMatom_get_a_num** returns the atomic number for atom. This function is valid only if **CHEMatom_init_user_data** has been previously called.

CHEMatom_set_a_num

```
C                                       Fortran
int                                     integer
CHEMatom_set_a_num(atom, a_num)         chematom_set_a_num(atom, a_num)

CHEMatom        *atom;                  integer atom
int               a_num;                  integer a_num
```

Given a valid CHEMatom object atom, **CHEMatom_set_a_num** assigns the atomic number for atom. A_num is constrained to be greater than or equal to 0. This function is valid only if **CHEMatom_init_user_data** has been previously called.

CHEMatom_get_hybrid

```
C                                       Fortran
int                                     integer
CHEMatom_get_hybrid(atom, hyb)          chematom_get_hybrid(atom, hyb)

    CHEMatom     *atom;                 integer atom
    int          *hyb;                  integer hyb
```

Given a valid CHEMatom object atom, **CHEMatom_get_hybrid** returns the hybridization type for atom. Hyb is constrained to be greater than or equal to 0. This function is valid only if **CHEMatom_init_user_data** has been previously called.

CHEMatom_set_hybrid

```
C                                       Fortran
int                                     integer
CHEMatom_set_hybrid(atom, hyb)          chematom_set_hybrid(atom, hyb)

CHEMatom     *atom;                     integer atom
int          *hyb;                      integer hyb
```

Given a valid CHEMatom object atom, **CHEMatom_set_hybrid** assigns the hybridization for atom. Hyb is constrained to be greater than or equal to 0. This function is valid only if **CHEMatom_init_user_data** has been previously called.

CHEMatom_get_weight

```
C                                       Fortran
int                                     integer
CHEMatom_get_weight(atom, weight)       chematom_get_weight(atom, weight)

CHEMatom          *atom;                integer atom
float           *weight;                real    weight
```

Given a valid CHEMatom object atom, **CHEMatom_get_weight** returns the weight for atom. Weight is constrained to be greater than or equal to 0. This function is valid only if **CHEMatom_init_user_data** has been previously called.

CHEMatom_set_weight

```
C                                       Fortran
int                                     integer
CHEMatom_set_weight(atom, w)            chematom_set_weight(atom, w)

CHEMatom          *atom;                integer atom
float           *weight;                real weight
```

Given a valid CHEMatom object atom, **CHEMatom_set_weight** assigns the weight for atom. Weight is constrained to be greater than or equal to 0. This function is valid only if **CHEMatom_init_user_data** has been previously called.

CHEMatom_get_charge

```
C                                       Fortran
int                                     integer
CHEMatom_get_charge(atom, charge)       chematom_get_charge(atom, charge)

CHEMatom          *atom;                integer atom
float           *charge;                real    charge
```

Given a valid CHEMatom object atom, **CHEMatom_get_charge** returns the charge for atom. This function is valid only if **CHEMatom_init_user_data** has been previously called.

CHEMatom_set_charge

```
C                                       Fortran
int                                     integer
CHEMatom_set_charge(atom, charge)       chematom_set_charge(atom, charge)

CHEMatom          *atom;                integer atom
float           *charge;                real    charge
```

Given a valid CHEMatom object atom, **CHEMatom_set_charge** assigns the charge for atom. This function is valid only if **CHEMatom_init_user_data** has been previously called.

Connectivity and Bond Interface Functions

The CHEMcandb object represents connectivity and bond pair information for a given atom. A CHEMatom object may or may not have an instance of a CHEMcandb list depending on the application's need. CHEMcandb objects

may be used to represent either a symmetric or asymmetric representation of CHEMatom connectivity.

The symmetrical representation requires that each atom's CHEMcandb object point half way to those that it connects with. For example, if CHEMatom A has a double bond to CHEMatom B, then CHEMatoms A and B must have instance of CHEMcandb that reference each other. They should both have the same bond type.

Asymmetric connectivity requires each atom to contain a CHEMcandb object that points all the way to the connected atom. For example, if CHEMatom A has a double bond to CHEMatom B, then CHEMatom A need only have an instance of CHEMcandb that references CHEMatom B. The connectivity should not be reciprocated by CHEMatom B. It is possible to have CHEMatoms that have no locally associated connectivity, yet be connected: their connectivity is defined by some other CHEMatom object.

The CDK assumes that all connectivity is asymmetric. There exist functions to aid in the conversion from the symmetric to asymmetric representations and vice versa. The CHEMcandb valid bond types include

- **CHEM_BOND_NONE**—no/undefined bond,
- **CHEM_BOND_SINGLE**—single bond,
- **CHEM_BOND_DOUBLE**—double bond,
- **CHEM_BOND_TRIPLE**—triple bond,
- **CHEM_BOND_HYDROGEN—**hydrogen bond,
- **CHEM_BOND_DISULFIDE**—disulfide bond and
- **CHEM_BOND_AROMATIC**—aromatic bond.

The CHEMcandb object contains:

```
int             conn;       internal atom offsets 0 to (natoms-1)
int             bond;       bond types.
```

CHEMcandb_alloc

```
C                                   Fortran
CHEMcandb*                          integer
CHEMcandb_alloc()                   chemcandb_alloc()
```

The **CHEMcandb_alloc** function returns a CHEMcandb object to the caller. All data is initialized to 0 and there are no ancillary objects. These Fortran and C functions will not return a CHEM error code; rather they will return NULL or 0 respectively.

CHEMcandb_free

```
C                                       Fortran
void
CHEMcandb_free(candb)                   chemcandb_free(candb)

CHEMcandb    *candb;                    integer candb
```

Given a valid CHEMcandb object cnb, **CHEMcandb_free** will traverse and free all members of the CHEMcandb list.

CHEMcandb_get

```
C                                       Fortran
int                                     integer
CHEMcandb_get(candb, conn, bond)        chemcandb_get(candb, conn, bond)

CHEMcandb    *candb;                    integer candb
int          *conn;                     integer conn
int          *bond;                     integer bond
```

Given a valid CHEMcandb object candb, **CHEMcandb_get** returns the offset of the atom connected, conn, and the bond type, bond. Conn is constrained to be greater than or equal to 0, but it is not constrained to be less than or equal to the number of atoms in the parent CHEMmolecule object. It is suggested that the application ensure that the conn offset is within the bounds of the total number of atoms for its CHEMmolecule object. The bond is constrained to be of the types, **CHEM_BOND_NONE, CHEM_BOND_SINGLE, CHEM_-BOND_DOUBLE, CHEM_BOND_TRIPLE, CHEM_BOND_HYDROGEN, CHEM_BOND_DISULFIDE** or **CHEM_BOND_AROMATIC**.

CHEMcandb_set

```
C                                       Fortran
int                                     integer
CHEMcandb_set(candb, conn, bond)        chemcandb_set(candb, conn, bond)

CHEMcandb    *candb;                    integer candb
int          conn;                      integer conn
int          bond;                      integer bond
```

Given a valid CHEMcandb object candb, **CHEMcandb_set** assigns the offset of the atom connected, conn, and the bond type, bond. Conn is constrained to be greater than or equal to 0, but it is not constrained to be less than or equal to the number of atoms in the parent CHEMmolecule object. It is suggested that the application ensure that the conn offset is within the bounds of the total number of atoms for its CHEMmolecule object. Bond is constrained to be

of the types, **CHEM_BOND_NONE, CHEM_BOND_SINGLE, CHEM_-BOND_DOUBLE, CHEM_BOND_TRIPLE, CHEM_BOND_HYDROGEN, CHEM_BOND_DISULFIDE** or **CHEM_BOND_AROMATIC**.

CHEMcandb_get_conn

```
C                                       Fortran
int                                     integer
CHEMcandb_get_conn(candb, conn)         chemcandb_get_conn(candb, conn)

CHEMcandb    *candb;                    integer candb
int          *conn;                     integer conn
```

Given a valid CHEMcandb object candb, **CHEMcandb_get_conn** assigns the offset of the atom connected, conn. Conn is constrained to be greater than or equal to 0, but it is not constrained to be less than or equal to the number of atoms in the parent CHEMmolecule object. It is suggested that the application ensure that the conn offset is within the bounds of the total number of atoms for its CHEMmolecule object.

CHEMcandb_set_conn

```
C                                       Fortran
int                                     integer
CHEMcandb_set_con(candb, conn)          chemcandb_set_conn(candb, conn)

CHEMcandb    *candb;                    integer candb
int          conn;                      integer conn
```

Given a valid CHEMcandb object candb, **CHEMcandb_set_conn** assigns the offset of the atom connected, conn. Conn is constrained to be greater than or equal to 0, but it is not constrained to be less than or equal to the number of atoms in the parent CHEMmolecule object. It is suggested that the application ensure that the conn offset is within the bounds of the total number of atoms for its CHEMmolecule object.

CHEMcandb_get_bond

```
C                                       Fortran
int                                     integer
CHEMcandb_get_bond(candb, bond)         chemcandb_get_bond(candb, bond)

CHEMcandb    *candb;                    integer candb
int          *bond;                     integer bond
```

Given a valid CHEMcandb object candb, **CHEMatom_get_bond**, returns the bond type, bond. Bond is constrained to be one of the types: **CHEM_-BOND_NONE, CHEM_BOND_SINGLE, CHEM_BOND_DOUBLE, CHEM_BOND_TRIPLE, CHEM_BOND_HYDROGEN, CHEM_BOND_-DISULFIDE** or **CHEM_BOND_AROMATIC**.

CHEMcandb_set_bond

```
C                                       Fortran
int                                     integer
CHEMcandb_set_bond(candb, bond)         chemcandb_set_bond(candb, bond)

CHEMcandb    *candb;                    integer candb
int          bond;                      integer bond
```

Given a valid CHEMcandb object candb, **CHEMatom_set_bond**, sets the bond type, bond. Bond is constrained to be one of the types: **CHEM_-BOND_NONE, CHEM_BOND_SINGLE, CHEM_BOND_DOUBLE, CHEM_BOND_TRIPLE, CHEM_BOND_HYDROGEN, CHEM_BOND_-DISULFIDE,** or **CHEM_BOND_AROMATIC**.

Chemunit (Substructure) Interface Functions

The CHEMchemunit object is used to represent chemical substructures. It may or may not be referenced by a CHEMmolecule object. The CHEM-chemunit object must be used in conjunction with a valid CHEMatom object list. The CHEMchemunit object is represented by and internal offset list. The components are listed below.

```
char                *name;      substructure unit name
int               unit_number;  substructure unit number
CHEMint_list        *atom;      internal atom offsets (0 to (natom-1))
int                 parent;     index of parent molecule
```

CHEMchemunit_alloc

```
C                                       Fortran
CHEMchemunit*                           integer
CHEMchemunit_alloc()                    chemchemunit_alloc()
```

The **CHEMchemunit_alloc** function returns a CHEMchemunit object to the caller. All data is initialized to 0, with all ancillary objects terminated. These Fortran and C functions will not return a CHEM error code; rather they will return 0 or NULL respectively.

CHEMchemunit_free

```
C                                       Fortran
void                                    integer
CHEMchemunit_free(cu)                   chemchemunit_free(cu)

CHEMchemunit  *cu;                      integer cu
```

Given a valid CHEMchemunit object cu, **CHEMchemunit_free** will traverse the list designated by cu and free all ancillary CHEMint_list lists.

CHEMchemunit_get

```
C                                       Fortran
int                                     integer
CHEMchemunit_get(cu, name,              chemchemunit_get(cu, name,
     number, int_l, parent)                 number, int_l, parent)

    CHEMchemunit *cu;                   integer       cu
    int          *number;               integer       number
    CHEMint_list **int_l;               integer       int_l
    int          *parent;               integer       parent
    char         **name;                character*256 name
```

Given a valid CHEMchemunit object, cu, **CHEMchemunit_get** will return all components of object cu. Any data that has not been previously set will be returned as 0, and any uninitialized ancillary object lists will be returned as 0 or NULL.

CHEMchemunit_set

```
C                                       Fortran
int                                     integer
CHEMchemunit_set(cu, name,              chemchemunit_set(cu, name,
     number, int_l, parent)                 number, int_l, parent)

    CHEMchemunit *cu;                   integer       cu
    int          number;                integer       number
    CHEMint_list *int_l;                integer       int_l
    int          parent;                integer       parent
    char         *name;                 character*256 name
```

Given a valid CHEMchemunit object, cu, **CHEMchemunit_get** will assign all components of object cu. Setting values to NULL or 0 respectively will ensure that ancillary lists are unused.

CHEMchemunit_get_number

```
C                                       Fortran
int                                     integer
CHEMchemunit_get_number(cu, number)     chemchemunit_get_number(cu, number)

CHEMchemunit *cu;                       integer       cu
int          *number;                   integer       number
```

Given a valid CHEMchemunit object cu, **CHEMchemunit_get_number** returns the unit number component of cu.

CHEMchemunit_set_number

```
C                                       Fortran
int                                     integer
CHEMchemunit_set_number(cu, number)     chemchemunit_set_number(cu, number)

CHEMchemunit *cu;                       integer       cu
int          number;                    integer       number
```

Given a valid CHEMchemunit object cu, **CHEMchemunit_set_number** assigns the unit number component of cu.

CHEMchemunit_get_name

```
C                                       Fortran
int                                     integer
CHEMchemunit_get_name(cu, name)         chemchemunit_get_name(cu, name)

CHEMchemunit *cu;                       integer       cu
char         **name;                    character*256 name
```

Given a valid CHEMchemunit object cu, **CHEMchemunit_get_name** will return the name component of cu.

CHEMchemunit_set_name

```
C                                       Fortran
int                                     integer
CHEMchemunit_set_name(cu, name)         chemchemunit_set_name(cu, name)

 CHEMchemunit *cu;                      integer       cu
 char         *name;                    character*256 name
```

Given a valid CHEMchemunit object cu, **CHEMchemunit_set_name**, will assign the name component of cu.

CHEMchemunit_get_parent

```
C                                       Fortran
int                                     integer
CHEMchemunit_get_parent(cu, parent)     chemchemunit_get_parent(cu, parent)

CHEMchemunit *cu;                       integer       cu
int          *parent ;                  integer       parent
```

Given a valid CHEMchemunit object cu, **CHEMchemunit_get_parent** will return an integer offset to the CHEMmolecule object that is parent of CHEMchemunit cu. Parent is constrained to be greater than or equal to 0, but it is not constrained to be less than or equal to the number of CHEMmolecules in the parent CHEMmolecule list. It is suggested that the application ensure that the parent offset is within the bounds of the total number of CHEMmolecules in the parent molecule list.

CHEMchemunit_set_parent

```
C                                       Fortran
int                                     integer
CHEMchemunit_set_parent(cu, parent)     chemchemunit_set_parent(cu, parent)

CHEMchemunit *cu;                       integer       cu
int          parent;                    integer       parent
```

Given a valid CHEMchemunit object cu, **CHEMchemunit_set_parent**, sets an integer offset to the CHEMmolecule that is parent of CHEMchemunit cu. Parent is constrained to be greater than or equal to 0, but it is not constrained to be less than or equal to the number of CHEMmolecules in the parent CHEMmolecule list. It is suggested that the application ensure that the parent offset is within the bounds of the total number of CHEMmolecules in the parent molecule list.

CHEMchemunit_get_int_list

```
C                                       Fortran
int                                     integer
CHEMchemunit_get_int_list(cu, int_list) chemchemunit_get_int_list(cu, int_list)

CHEMchemunit *cu;                       integer       cu
CHEMint_list **int_list;                integer       int_list
```

Given a valid CHEMchemunit object cu, **CHEMchemunit_get_int_list**, returns the root of the list of internal offsets. This list specifies which atoms of the parent molecule object belong to this chemunit object.

CHEMchemunit_set_int_list

```
C                                       Fortran
int                                     integer
CHEMchemunit_set_int_list(cu, int_list) chemchemunit_set_int_list(cu, int_list)

CHEMchemunit *cu;                       integer       cu
CHEMint_list *int_list;                 integer       int_list
```

Given a valid CHEMchemunit object cu, **CHEMchemunit_get_int_list**, assigns the root of the list of internal offsets. This list specifies which atoms of the parent molecule object belong to this chemunit object.

Internal List Interface Functions

The CHEMint_list object is used to represent an internal offset list for a set of atoms. Each valid CHEMchemunit object should instance a CHEMint_list object that represents a valid list of atom offsets.

```
int                     off;    offset
```

CHEMint_list_alloc

```
C                                       Fortran
CHEMint_list*                           integer
CHEMint_list_alloc()                    chemint_list_alloc()
```

The **CHEMint_list_alloc** function returns a CHEMint_list object to the caller. All data is initialized to 0 and there are no ancillary objects. These Fortran and C functions will not return a CHEM error code; rather they will return 0 or NULL respectively.

CHEMint_list_free

```
C                                       Fortran
void
CHEMint_list_free(int_list)             chemint_list_free(int_list)

CHEMint_list    *int_list;              integer int_list
```

Given a valid CHEMint_list object, int_lst, **CHEMint_list_free** will traverse and free all members of the CHEMint_list list.

CHEMint_list_get_off

```
C                                       Fortran
int                                     integer
CHEMint_list_get_off(int_list, off)     chemint_list_get_off(int_list, off)

CHEMint_list *int_list;                 integer       int_list
int          *off;                      integer       off
```

Given a valid CHEMint_list object int_list, **CHEMint_list_set_off**, will assign the offset component. Offset is constrained to be greater than or equal to 0, but it is not constrained to be less than or equal to the number of CHEMatoms of the parent CHEMmolecule. It is the application's responsibiliy to ensure the legality of the offset value.

CHEMint_list_set_off

```
C                                       Fortran
int                                     integer
CHEMint_list_set_off(int_list, off)     chemint_list_set_off(int_list, off)

CHEMint_list   *int_list;               integer       int_list
int            off;                     integer       off
```

Given a valid CHEMint_list object int_list, **CHEMint_list_set_off**, will assign the offset component. Offset is constrained to be greater than or equal to 0, and should be less than or equal to the number of CHEMatoms of the parent CHEMmolecule. It is the application's responsibility to ensure the legality of the offset value.

Quantum Interface Functions

The CHEMquantum object is used to represent the quantum information in the molecule data type. There need not be any corresponding CHEMatom or CHEMchemunit objects in order to complete CHEMquantum definition. The User Data component of the CHEMquantum object is provided for application use if desired.

```
char              *name;     name of CHEM quantum object
int              nbasis;     number of basis functions
int                 ich;     overall charge of molecule
int                 mul;     multiplicity
int                  ne;     number of electrons
int                  na;     number of alpha spin electrons
```

```
int                   nb;     number of beta spin electrons
int               scftype;    type of SCF (self-consistent field)
                              calculation (RHF for example).
int              corrtype;    type of higher-order calculation
                              (MP2 for example)
int                 npage;    number of CHEMq_pages
int                nshell;    number of CHEMshells
int                ngauss;    total number of CHEMgauss
CHEMshell          *shell;    pointer to CHEMshell object list
CHEMq_page            *pl;    pointer to CHEMq_page object list
                              dimensioned nbasis*mbasis; may hold
                              molecular orbital coefficients or other
                              data
char*           User_Data;    for User Data
```

CHEMquantum_alloc

```
C                                   Fortran
CHEMquantum*                        integer
CHEMquantum_alloc()                 chemquantum_alloc()
```

The **CHEMquantum_alloc** function returns a CHEMquantum object to the caller. All data is initialized to 0, with no ancillary objects. These Fortran and C functions will not return a CHEM error code; rather they will return 0 or NULL respectively.

CHEMquantum_free

```
C                                   Fortran
void                                integer
CHEMquantum_free(quant)             chemquantum_free(quant)

CHEMquantum     *quant;             integer   quant;
```

Given a valid CHEMquantum object, quant, **CHEMquantum_free** will traverse the list designated by quant and free all ancillary objects.

CHEMquantum_get

```
C                                   Fortran
int                                 integer
CHEMquantum_get(quant, name, nbasis,  CHEMquantum_get(quant, name, nbasis,
   ich, mul, ne, na, scftype, corrtype,   ich, mul, ne, na, scftype, corrtype,
   np, ns, ng, shell, page, udat)         np, ns, ng, shell, page, udat)

CHEMquantum *quant;                 integer       quant
char        **name;                 character*256 name
int         *nbasis;                integer       nbasis
```

```
int         *ich;                    integer       ich
int         *mul;                    integer       mul
int         *ne;                     integer       ne
int         *na;                     integer       na
int         *nb;                     integer       nb
int         *scftype;                integer       scftype
int         *corrtype;               integer       corrtype
int         *np;                     integer       np
int         *ns;                     integer       ns
int         *ng;                     integer       ng
CHEMshell   **shell;                 integer       shell
CHEMq_page  **page;                  integer       page
char        **udat;                  integer       udat
```

Given a valid CHEMquantum object, quant, **CHEMquantum_set** will return all components of the specified quantum object. Any data that has not been previously set will be returned as 0, and any uninitialized ancillary object lists will be terminated.

CHEMquantum_set

```
C                                    Fortran
int                                  integer
CHEMquantum_set(quant, name, nbasis,    CHEMquantum_set(quant, name, nbasis,
   ich, mul, ne, na, scftype, corrtype,    ich, mul, ne, na, scftype, corrtype,
   np, ns, ng, shell, page, udat)          np, ns, ng, shell, page, udat)

CHEMquantum *quant;                  integer       quant
char        *name;                   character*256 name
int         nbasis;                  integer       nbasis
int         ich;                     integer       ich
int         mul;                     integer       mul
int         ne;                      integer       ne
int         na;                      integer       na
int         nb;                      integer       nb
int         scftype;                 integer       scftype
int         corrtype;                integer       corrtype
int         np;                      integer       np
int         ns;                      integer       ns
int         ng;                      integer       ng
CHEMshell   *shell;                  integer       shell
CHEMq_page  *page;                   integer       page
char        *udat;                   integer       udat
```

Given a valid CHEMquantum object, quant, **CHEMquantum_get** will assign all components of the specified quantum object. Any data that has not been previously set will be returned as 0, and any uninitialized ancillary object lists will be terminated. Care should be taken to ensure the user data component of the CHEMquantum object is created using the proper utilities. Refer to the *AVS Developer's Guide* for more information regarding the use of user data.

CHEMquantum_get_name

```
C                                       Fortran
int                                     integer
CHEMquantum_get_name(quant, name)       chemquantum_get_name(quant, name)

CHEMquantum *quant;                     integer       quant
char        **name;                     character*256 name
```

Given a valid CHEMquantum object, quant, **CHEMquantum_get_name** will return the specified quantum objects name.

CHEMquantum_set_name

```
C                                       Fortran
int                                     integer
CHEMquantum_set_name(quant, name)       chemquantum_set_name(quant, name)

CHEMquantum *quant;                     integer       quant
char        *name;                      character*256 name
```

Given a valid CHEMquantum object, quant, **CHEMquantum_set_name** will assign the name of the specified quantum object.

CHEMquantum_get_nbasis

```
C                                       Fortran
int                                     integer
CHEMquantum_get_nbasis(quant,nbasis)    chemquantum_get_nbasis(quant,nbasis)

CHEMquantum *quant;                     integer    quant
int         *nbasis;                    integer    nbasis
```

Given a valid CHEMquantum object, quant, **CHEMquantum_get_nbasis** will return the value of number of basis functions, nbasis. Nbasis is constrained to be greater than or equal to 0.

CHEMquantum_set_nbasis

```
C                                       Fortran
int                                     integer
CHEMquantum_set_nbasis(quant, nbasis)   chemquantum_set_nbasis(quant, nbasis)

CHEMquantum *quant;                     integer     quant
int          nbasis;                    integer     nbasis
```

Given a valid CHEMquantum object, quant, **CHEMquantum_set_nbasis** will assign the number of basis functions. Nbasis is constrained to be greater than or equal to 0.

CHEMquantum_get_user_data

```
C                                       Fortran
int                                     integer
CHEMquantum_get_user_data(quant, udat)  chemquantum_get_user_data(quant, udat)

CHEMquantum *quant;                     integer     quant
char        **udat;                     integer     udat
```

Given a valid CHEMquantum object, quant, **CHEMquantum_get_user_data** will return the user data component, udat. Care should be taken to ensure the user_data component of the CHEMquantum object is created using the proper utilities. Refer to the *AVS Developer's Guide* for more information regarding the use of user data.

CHEMquantum_set_user_data

```
C                                       Fortran
int                                     integer
CHEMquantum_set_user_data(quant, udat)  chemquantum_set_user_data(quant, udat)

CHEMquantum *quant;                     integer     quant
char        *udat;                      integer     udat
```

Given a valid CHEMquantum object, quant, **CHEMquantum_set_user_data** will return the quantum's user_data, udat. Care should be taken to ensure the user data component of the CHEMquantum object is created using the proper utilities. Refer to the *AVS Developer's Guide* for more information regarding the use of user data.

CHEMquantum_get_ich

```
C                                   Fortran
int                                 integer
CHEMquantum_get_ich(quant, ich)     chemquantum_get_ich(quant, ich)

CHEMquantum    *quant;              integer     quant
int             *ich;               integer     ich
```

Given a valid CHEMquantum object, quant, **CHEMquantum_get_ich** will return the overall charge of the molecule, ich.

CHEMquantum_set_ich

```
C                                       Fortran
int                                     integer
CHEMquantum_set_ich(quant, ich)         chemquantum_set_ich(quant, ich)

CHEMquantum *quant;                     integer     quant
int         ich;                        integer     ich
```

Given a valid CHEMquantum object, quant, **CHEMquantum_set_ich** will assign the overall charge of the molecule, ich.

CHEMquantum_get_mul

```
C                                       Fortran
int                                     integer
CHEMquantum_get_mul(quant, mul)         chemquantum_get_mul(quant, mul)

CHEMquantum    *quant;                  integer     quant
int            *mul;                    integer     mul
```

Given a valid CHEMquantum object, quant, **CHEMquantum_get_mul** will return the multiplicity of the quantum object. The multiplicity is constrained to be greater than or equal to 0.

CHEMquantum_set_mul

```
C                                       Fortran
int                                     integer
CHEMquantum_set_mul(quant, mul)         chemquantum_set_mul(quant, mul)

CHEMquantum *quant;                     integer     quant
int          mul;                       integer     mul
```

Given a valid CHEMquantum object, quant, **CHEMquantumset_mu**l will return the multiplicity of the quantum object. The multiplicity is constrained to be greater than or equal to 0.

CHEMquantum_get_ne

```
C                                       Fortran
int                                     integer
CHEMquantum_get_ne(quant, ne)           chemquantum_get_ne(quant, ne)

CHEMquantum    *quantum;                integer     quant
```

```
int            *ne;                    integer     ne
```

Given a valid CHEMquantum object, quant, **CHEMquantum_get_ne** will return the number of electrons, ne. Ne is constrained to be greater than or equal to 0.

CHEMquantum_set_ne

```
C                                      Fortran
int                                    integer
CHEMquantum_set_ne(quant, ne)          chemquantum_set_ne(quant, ne)

CHEMquantum *quant;                    integer     quant
int         ne;                        integer     ne
```

Given a valid CHEMquantum object, quant, **CHEMquantum_set_ne** will assign the number of electrons, ne. Ne is constrained to be greater than or equal to 0.

CHEMquantum_get_na

```
C                                      Fortran
int                                    integer
CHEMquantum_get_na(quant, na)          chemquantum_get_na(quant, na)

CHEMquantum  *quant;                   integer     quant
int          *na;                      integer     na
```

Given a valid CHEMquantum object, quant, **CHEMquantum_get_na** will return the number of alpha spin electrons, na. Na is constrained to be greater than or equal to 0.

CHEMquantum_set_na

```
C                                      Fortran
int                                    integer
CHEMquantum_set_na(quant, na)          chemquantum_set_na(quant, na)

 CHEMquantum *quant;                   integer     quant
 int         na;                       integer     na
```

Given a valid CHEMquantum object, quant, **CHEMquantum_set_na** will assign the number of alpha spin electrons, na. Na is constrained to be greater than or equal to 0.

CHEMquantum_get_nb

```
C                                       Fortran
int                                     integer
CHEMquantum_get_nb(quant, nb)           chemquantum_get_nb(quant, nb)

CHEMquantum    *quant;                  integer     quant
int            *nb;                     integer     nb
```

Given a valid CHEMquantum object, quant, **CHEMquantum_get_nb** will return the number of beta spin electrons, nb. Nb is constrained to be greater than or equal to 0.

CHEMquantum_set_nb

```
C                                       Fortran
int                                     integer
CHEMquantum_set_nb(quant, nb)           chemquantum_set_nb(quant, nb)

CHEMquantum *quant;                     integer     quant
int          nb;                        integer     nb
```

Given a valid CHEMquantum object, quant, **CHEMquantum_set_nb** will assign the number of beta spin electrons, nb. Nb is constrained to be greater than or equal to 0.

CHEMquantum_get_scftype

```
C                                       Fortran
int                                     integer
CHEMquantum_get_scftype(quant, scftype) chemquantum_get_scftype(quant, scftype)

CHEMquantum   *quant;                   integer     quant
int           *scftype;                 integer     scftype
```

Given a valid CHEMquantum object, quant, **CHEMquantum_set_scftype** will return the scftype. Scftype is constrained to be greater than or equal to 0.

CHEMquantum_set_scftype

```
C                                       Fortran
int                                     integer
CHEMquantum_set_scftype(quant, scftype) chemquantum_set_scftype(quant, scftype)

CHEMquantum *quant;                     integer     quant
```

```
int         scftype;                    integer     scftype
```

Given a valid CHEMquantum object, quant, **CHEMquantum_set_scftype** will assign the scftype. Scftype is constrained to be greater than or equal to 0.

CHEMquantum_get_corrtype

```
C                                       Fortran
int                                     integer
CHEMquantum_get_corrtype(quant, cortyp) chemquantum_get_corrtype(quant, cortyp)

CHEMquantum   *quant;                   integer     quant
int           *cortyp;                  integer     cortyp
```

Given a valid CHEMquantum object, quant, **CHEMquantum_get_scftype** will return the correlation type, cortyp. Cortyp is constrained to be greater than or equal to 0.

CHEMquantum_set_corrtype

```
C                                       Fortran
int                                     integer
CHEMquantum_set_corrtype(quant, cortyp) chemquantum_set_corrtype(quant, cortyp)

CHEMquantum *quant;                     integer     quant
int         cortyp;                     integer     cortyp
```

Given a valid CHEMquantum object, quant, **CHEMquantum_set_nb** will assign the correlation type, cortyp. Cortyp is constrained to be greater than or equal to 0.

CHEMquantum_get_npage

```
C                                       Fortran
int                                     integer
CHEMquantum_get_npage(quant, npage)     chemquantum_get_npage(quant, npage)

CHEMquantum   *quant;                   integer     quant
int           *npage;                   integer     npage
```

Given a valid CHEMquantum object, quant, **CHEMquantum_get_npage** will return the number of q_pages, npage, for this quantum representation. Npage is constrained to be greater than or equal to 0.

CHEMquantum_set_npage

```
C                                       Fortran
int                                     integer
CHEMquantum_set_npage(quant, npage)     chemquantum_set_npage(quant, npage)

CHEMquantum  *quant;                    integer     quant
int          npage;                     integer     npage
```

Given a valid CHEMquantum object, quant, **CHEMquantum_set_npage** will set the number of q_pages, npage, defined for this quantum representation. Npage is constrained to be greater than or equal to 0.

CHEMquantum_get_nshell

```
C                                       Fortran
int                                     integer
CHEMquantum_get_nshell(quant, nshell)   chemquantum_get_nshell(quant, nshell)

CHEMquantum    *quant;                  integer     quant
int            *nshell;                 integer     nshell
```

Given a valid CHEMquantum object, quant, **CHEMquantum_get_shell** will return the number of shells, nshell, defined for this quantum representation. Nshell is constrained to be greater than or equal to 0.

CHEMquantum_set_nshell

```
C                                       Fortran
int                                     integer
CHEMquantum_set_nshell(quant, nshell)   chemquantum_set_nshell(quant, nshell)

CHEMquantum *quant;                     integer     quant
int          nshell;                    integer     nshell
```

Given a valid CHEMquantum object, quant, **CHEMquantum_set_nshell** will assign the number of shells defined for the this quantum representation. Nshell is constrained to be greater than or equal to 0.

CHEMquantum_get_ngauss

```
C                                       Fortran
int                                     integer
CHEMquantum_get_ngauss(quant, ngauss)   chemquantum_get_ngauss(quant, ngauss)

CHEMquantum   *quant;                   integer     quant
```

```
int          *ngauss                    integer     ngauss
```

Given a valid CHEMquantum object, quant, **CHEMquantum_get_ngauss** will return the number of gaussians defined for the this quantum representation. Ngauss is constrained to be greater than or equal to 0.

CHEMquantum_set_ngauss

```
C                                       Fortran
int                                     integer
CHEMquantum_set_ngauss(quant, ngauss)   chemquantum_set_ngauss(quant, ngauss)

CHEMquantum *quant;                     integer     quant
int         ngauss;                     integer     ngauss
```

Given a valid CHEMquantum object, quant, **CHEMquantum_set_ngauss** will assign the number of gaussians defined for the this quantum representation. Ngauss is constrained to be greater than or equal to 0.

CHEMquantum_get_shell

```
C                                       Fortran
int                                     integer
CHEMquantum_get_shell(quant, shell)     chemquantum_get_shell(quant, shell)

CHEMquantum    *quant;                  integer     quant
CHEMshell      **shell;                 integer     shell
```

Given a valid CHEMquantum object, quant, **CHEMquantum_get_shell** will return the root of the CHEMshell list, shell, attached to this quantum representation.

CHEMquantum_set_shell

```
C                                       Fortran
int                                     integer
CHEMquantum_set_shell(quant, shell)     chemquantum_set_shell(quant, shell)

 CHEMquantum *quant;                    integer     quant
 CHEMshell   *shell;                    integer     shell
```

Given a valid CHEMquantum object, quant, **CHEMquantum_set_shell** will assign the root of the CHEMshell list, shell.

CHEMquantum_get_q_page

```
C                                       Fortran
int                                     integer
CHEMquantum_get_q_page(quant, q_page)   chemquantum_get_q_page(quant, q_page)

CHEMquantum    *quant;                  integer     quant
CHEMq_page     **q_page;                integer     q_page
```

Given a valid CHEMquantum object, quant, **CHEMquantum_get_q_page** will return the root of the CHEMq_page list, q_page.

CHEMquantum_set_q_page

```
C                                       Fortran
int                                     integer
CHEMquantum_set_q_page(quant, qp)       chemquantum_set_q_page(quant, q)

CHEMquantum   *quant;                   integer     quant
CHEMq_page    *qp;                      integer     qp
```

Given a valid CHEMquantum object, quant, **CHEMquantum_set_q_page**, will set the root of the CHEMq_page list, q_page.

Quantum Matrix Interface Functions

The CHEMq_page object represents the coefficient matrices for quantum representation. The CHEMq_page list is instanced by the CHEMquantum object. Each coefficient matrix is dimensioned mbasis by nbasis and is named. There are page, row and column operators for extracting and setting information in the page.

```
char*              name;      name of this page
int              mbasis;      mbasis row
int              nbasis;      nbasis col
double            *page;      dimensioned [mbasis*nbasis]
```

CHEMq_page_alloc

```
C                                       Fortran
CHEMq_page*                             integer
CHEMq_page_alloc()                      chemq_page_alloc()
```

The **CHEMq_page_alloc** function returns a CHEMq_page object to the caller. All data is initialized to 0, with no ancillary objects. These Fortran and C

functions will not return a CHEM error code; rather they will return 0 or NULL respectively.

CHEMq_page_free

```
C                                       Fortran
void                                    integer
CHEMq_page_free(q_pag)                  chemq_page_free(q_pag)

CHEMq_page    *q_pag;                   integer   q_pag;
```

Given a valid CHEMq_page object q_pag, **CHEMq_page_free** will traverse the list by q_pag and free all ancillary objects of q_pag.

CHEMq_page_get

```
C                                       Fortran
int                                     integer
CHEMq_page_get(q_pag, name,             chemq_page_get(q_pag, name,
    m, n, mtx)                                          m, n, mtx)

CHEMq_page   *q_page;                   integer       q_pag
char         **name;                    character*256 name
int          *m;                        integer       m
int          *n;                        integer       n
double       **mtx;                     real*8        mtx()
```

Given a valid CHEMq_page*, q_page, **CHEMq_page_get** will return all components of q_page.

CHEMq_page_set

```
C                                       Fortran
int                                     integer
CHEMq_page_set(q_pag, name,             chemq_page_set(q_pag, name,
    q_page, n, mtx)                                 q_page, n, mtx)

CHEMq_page   *q_pag;                    integer       q_pag
char         *name;                     character*256 name
int          m;                         integer       m
int          n;                         integer       n
double       *mtx;                      integer       mtx()
```

Given a valid CHEMq_page*, q_page, **CHEMq_page_set** will assign all components of q_page according to the passed parameters.

CHEMq_page_init_page

```
C                                       Fortran
int                                     integer
CHEMq_page_init_page(q_pag,  m, n )     chemq_page_init_page(q_pag, m, n)

CHEMq_page   *q_pag;                    integer       q_pag
int           m;                        integer       m
int           n;                        integer       n
```

Given a valid CHEMq_page*, q_page, **CHEMq_page_init_page** will allocate the 2D matrix of double values integral to the CHEMq_page structure. The matrix created m rows by n cols and is initialized to 0.

CHEMq_page_get_ij

```
C                                       Fortran
int                                     integer
CHEMq_page_get_ij(q_pag, i, j, v)       chemq_page_get_ij(q_pag, i, j, v)

CHEMq_page *q_pag;                      integer       q_pag
int        i;                           integer       i
int        j;                           integer       j
double     *v;                          real*8        v
```

Given a valid CHEMq_page*, q_page, **CHEMq_page_get_ij** will return the value in the *i*th row and *j*th col of the 2D matrix. I and j are checked to ensure that they are within the bounds of the matrix.

CHEMq_page_set_ij

```
C                                       Fortran
int                                     integer
CHEMq_page_set_ij(q_pag, i, j, v)       chemq_page_set_ij(q_pag, i, j, v)

CHEMq_page *q_pag;                      integer       q_pag
int       i;                            integer       i
int       j;                            integer       j
double    v;                            real*8        v
```

Given a valid CHEMq_page*, q_page, **CHEMq_page_set_ij** will assign the value in the *i*th row and *j*th col of the 2D matrix to v. I and j are checked to ensure that they are within the bounds of the matrix.

CHEMq_page_get_col

```
C                                       Fortran
int                                     integer
CHEMq_page_get_col(q_pag, j, v)         chemq_page_get_col(q_pag, j, v)

CHEMq_page *q_pag;                      integer       q_pag
int       j;                            integer       j
double    **v;                          real*8        v()
```

Given a valid CHEMq_page*, q_page, **CHEMq_page_get_col** will fill the location, v, with the *j*th col of the 2D matrix. J is checked to ensure that it is within the bounds of the matrix. It is the application's responsibility to ensure that v is large enough to hold the column.

CHEMq_page_set_col

```
C                                       Fortran
int                                     integer
CHEMq_page_set_col(q_pag, j, v)         chemq_page_set_col(q_pag, j, v)

CHEMq_page *q_pag;                      integer       q_pag
int j;                                  integer       j
double *v;                              real*8        v()
```

Given a valid CHEMq_page*, q_page, **CHEMq_page_set_col** will assign the *j*th col of the 2D matrix with the contents of v. J is checked to ensure that it is within the bounds of the matrix. It is the application's responsibility to ensure that v is large enough to hold the column and that the values are valid.

CHEMq_page_get_row

```
C                                       Fortran
int                                     integer
CHEMq_page_get_row(q_pag, i, v)         chemq_page_get_row(q_pag, i, v)

CHEMq_page *q_pag;                      integer     q_pag
int i;                                  integer     i
double **v;                             real*8      v()
```

Given the a valid CHEMq_page*, q_page, **CHEMq_page_get_row** will fill the location, v, with the *i*th row of the 2D matrix. I is checked to ensure that it is within the bounds of the matrix. It is the application's responsibility to ensure that v is large enough to hold the row.

CHEMq_page_set_row

```
C                                       Fortran
int                                     integer
CHEMq_page_set_row(q_pag, i, v)         chemq_page_set_row(q_pag, i, v)

CHEMq_page *q_pag;                      integer     q_pag
int i;                                  integer     i
double **v;                             real*8      v()
```

Given the a valid CHEMq_page*, q_page, **CHEMq_page_set_row** will assign the *i*th col of the 2D matrix with the contents of v. I is checked to ensure that it is within the bounds of the matrix. It is the application's responsibility to ensure that v is large enough to hold the row and the values are valid.

CHEMq_page_get_name

```
C                                       Fortran
int                                     integer
CHEMq_page_get_name(q_pag, name)        chemq_page_get_name(q_pag, name)

CHEMq_page *q_page;                     integer       q_pag
char       **name;                      character*256 name
```

Given a valid CHEMq_page*, q_page, **CHEMq_page_get_name** will return the specified CHEMq_page name.

CHEMq_page_set_name

```
C                                       Fortran
int                                     integer
CHEMq_page_set_name(q_pag, name)        chemq_page_set_name(q_pag, name)

CHEMq_page *q_pag;                      integer     q_pag
char  *name;                            character*256 name
```

Given a valid CHEMq_page*, q_page, **CHEMq_page_set_name** will assign the name of the specified CHEMq_page.

Shell Interface Functions

The CHEMshell object represents a basis set for a given quantum object. The CHEMshell object is instanced by CHEMquantum object.

```
char           *name;     shell name
int           kstart;     starting location of the shell within
                          the complete list of primitive gaussians
int            katom;     number of the atom on which the shell resides
int            ktype;     integer description of the type of shell: S, P,
                          D, SP, F... This value is application specific
int              kng;     number of primitive gaussians for this shell
int             kloc;     In the list of basis functions for this molecule,
                          the first function which this shell describes
int             kmin;     starting atomic orbital type
int             kmax;     ending atomic orbital type
CHEMgauss     *gauss;     pointer the CHEMgauss object list.
```

CHEMshell_alloc

```
C                                   Fortran
CHEMshell*                          integer
CHEMshell_alloc()                   chemshell_alloc()
```

The **CHEMshell_alloc** function returns a CHEMshell object to the caller. All data is initialized to 0, with no ancillary objects. These Fortran and C functions will not return a CHEM error code; rather they will return 0 or NULL respectively.

CHEMshell_free

```
C                                   Fortran
void                                integer
CHEMshell_free(shell)               chemshell_free(shell)

CHEMshell     *shell;               integer   shell
```

Given a valid CHEMshell object, shell, **CHEMshell_free** will traverse the list by shell and free all ancillary objects of shell.

CHEMshell_get

```
C                                   Fortran
integer                             integer
CHEMshell_get(shell, name, kstart,  chemshell_get(shell, name, kstart,
    kshell, ktype, kng,                kshell, ktype, kng,
    kloc, kmin, kmax, gauss)           kloc, kmin, kmax, gauss)

CHEMshell  *shell;                  integer       shell
char       **name;                  character*256 name
int        *kstart;                 integer       kstart
int        *kshell;                 integer       kshell
int        *ktype;                  integer       ktype
```

```
int        *kng;                    integer       kng
int        *kloc;                   integer       kloc
int        *kmin;                   integer       kmin
int        *kmax;                   integer       kmax
CHEMgauss  **gauss;                 integer       gauss
```

Given a valid CHEMshell object, shell, **CHEMshell_ge**t will return all the values of shell.

CHEMshell_set

```
C                                   Fortran
integer                             integer
CHEMshell_set(shell, name, kstart,  chemshell_set(shell, name, kstart,
    kshell, ktype, kng,                kshell, ktype, kng,
    kloc, kmin, kmax, gauss)           kloc, kmin, kmax, gauss)

CHEMshell  *shell;                  integer       shell
char       *name;                   character*256 name
int        kstart;                  integer       kstart
int        kshell;                  integer       kshell
int        ktype;                   integer       ktype
int        kng;                     integer       kng
int        kloc;                    integer       kloc
int        kmin;                    integer       kmin
int        kmax;                    integer       kmax
CHEMgauss  *gauss;                  integer       gauss
```

Given a valid CHEMshell object, shell, **CHEMshell_set** will assign all components for shell.

CHEMshell_get_name

```
C                                   Fortran
integer                             integer
CHEMshell_get_name(shell, name)     chemshell_get_name(shell, name)

CHEMshell *shell;                   integer       shell
char    **name;                     character*256 name
```

Given a valid CHEMshell object, shell, **CHEMshell_get_name** will return the shell's name component.

CHEMshell_set_name

```
C                                   Fortran
integer                             integer
```

```
CHEMshell_set_name(shell, name)          chemshell_set_name(shell, name)

CHEMshell *shell;                        integer       shell
char      *name;                         character*256 name
```

Given a valid CHEMshell object, shell, **CHEMshell_set_name** will assign the name of shell's name component.

CHEMshell_get_katom

```
C                                        Fortran
integer                                  integer
CHEMshell_get_katom(shell, katom)        chemshell_get_katom(shell, katom)

CHEMshell *shell;                        integer       shell
int       *katom;                        integer       katom
```

Given a valid CHEMshell object, shell, **CHEMshell_get_katom** will return the internal offset of the atom on which this shell resides. Katom is constrained to be greater than or equal to 0, but it is not constrained to be less than or equal to the number of CHEMatoms in any CHEMmolecule list. It is suggested that the application ensure that the katom offset is within the bounds of the total number of CHEMatoms in the appropriate CHEMmolecule object.

CHEMshell_set_katom

```
C                                        Fortran
integer                                  integer
CHEMshell_set_katom(shell, katom)        chemshell_set_katom(shell, katom)

CHEMshell *shell;                        integer       shell
int        katom;                        integer       katom
```

Given a valid CHEMshell object, shell, **CHEMshell_set_katom** will set the internal offset of the atom on which this shell resides. Katom is constrained to be greater than or equal to 0, but it is not constrained to be less than or equal to the number of CHEMatoms in any CHEMmolecule list. It is suggested that the application ensure that the katom offset is within the bounds of the total number of CHEMatoms in the appropriate CHEMmolecule object.

CHEMshell_get_ktype

```
C                                        Fortran
integer                                  integer
CHEMshell_get_ktype(shell, ktype)        chemshell_get_ktype(shell, ktype)
```

```
CHEMshell *shell;                    integer       shell
int       *ktype;                    integer       ktype
```

Given a valid CHEMshell object, shell, **CHEMshell_get_ktype** will return the shell type. Ktype is constrained to be greater than or equal to 0.

CHEMshell_set_ktype

```
C                                    Fortran
integer                              integer
CHEMshell_set_ktype(shell, ktype)    chemshell_set_ktype(shell, ktype)

CHEMshell *shell;                    integer       shell
int       ktype;                     integer       ktype
```

Given a valid CHEMshell object, shell, C**HEMshell_set_ktyp**e will assign the shell type. Ktype is constrained to be greater than or equal to 0.

CHEMshell_get_kng

```
C                                    Fortran
integer                              integer
CHEMshell_get_kng(shell, kng)        chemshell_get_kng(shell, kng)

CHEMshell *shell;                    integer       shell
int       *kng;                      integer       kng
```

Given a valid CHEMshell object, shell, **CHEMshell_get_kng** will return the number of this shells gaussian references. Kng is constrained to be greater than or equal to 0, but to be less than the parent quantum object's number of gaussians.

CHEMshell_set_kng

```
C                                    Fortran
integer                              integer
CHEMshell_set_kng(shell, kng)        chemshell_set_kng(shell, kng)

CHEMshell *shell;                    integer       shell
int       kng;                       integer       kng
```

Given a valid CHEMshell object, shell, **CHEMshell_set_kng** will assign the number of this shells gaussian references. Kng is constrained to be greater

than or equal to 0, but to be less than the parent quantum object's number of gaussians.

CHEMshell_get_kstart

```
C                                       Fortran
integer                                 integer
CHEMshell_get_kstart(shell, kstart)     chemshell_get_kstart(shell, kstart)

CHEMshell *shell;                       integer       shell
int       *kstart;                      integer       kstart
```

Given a valid CHEMshell object, shell, **CHEMshell_get_kstart** will return the starting location for the shell within the gaussian list. Kstart is constrained to be greater than or equal to 0, but to be less than the parent quantum object's number of gaussians.

CHEMshell_set_kstart

```
C                                       Fortran
integer                                 integer
CHEMshell_set_kstart(shell, kstart)     chemshell_set_kstart(shell, kstart)

CHEMshell *shell;                       integer       shell
int        kstart;                      integer       kstart
```

Given a valid CHEMshell object, shell, **CHEMshell_set_kstart** will set the starting location for the shell within the gaussian list. Kstart is constrained to be greater than or equal to 0, but to be less than the parent quantum object's number of gaussians.

CHEMshell_get_kloc

```
C                                       Fortran
integer                                 integer
CHEMshell_get_kloc(shell, kloc)         chemshell_get_kloc(shell, kloc)

CHEMshell *shell;                       integer       shell
int       *kloc;                        integer       kloc
```

Given a valid CHEMshell object, shell, **CHEMshell_get_kloc** will return the value of the starting atomic orbital basis function. Kloc is constrained to be greater than or equal to 0.

CHEMshell_set_kloc

```
C                                       Fortran
integer                                 integer
CHEMshell_set_kloc(shell, kloc)         chemshell_set_kloc(shell, kloc)

CHEMshell *shell;                       integer       shell
int        kloc;                        integer       kloc
```

Given a valid CHEMshell object, shell, **CHEMshell_get_kloc** will assign the value of the starting atomic orbital basis function. Kloc is constrained to be greater than or equal to 0.

CHEMshell_get_kmin

```
C                                       Fortran
integer                                 integer
CHEMshell_get_kmin(shell, kmin)         chemshell_get_kmin(shell, kmin)

CHEMshell *shell;                       integer       shell
int       *kmin;                        integer       kmin
```

Given a valid CHEMshell object, shell, **CHEMshell_get_kmin** will return the starting atomic orbital basis function. Kmin is constrained to be greater than or equal to 0.

CHEMshell_set_kmin

```
C                                       Fortran
integer                                 integer
CHEMshell_set_kmin(shell, kmin)         chemshell_set_kmin(shell, kmin)

CHEMshell *shell;                       integer       shell
int       kmin;                         integer       kmin
```

Given a valid CHEMshell object, shell, **CHEMshell_set_kmin** will set the starting atomic orbital basis function. Kmin is constrained to be greater than or equal to 0.

CHEMshell_get_kmax

```
C                                       Fortran
integer                                 integer
CHEMshell_get_kmax(shell, kmax)         chemshell_get_kmax(shell, kmax)

CHEMshell *shell;                       integer       shell
```

```
int       kmax;                          integer       kmax
```

Given a valid CHEMshell object, shell, **CHEMshell_get_kmax** will return the ending atomic orbital basis function. Kmax is constrained to be greater than or equal to 0.

CHEMshell_set_kmax

```
C                                        Fortran
integer                                  integer
CHEMshell_set_kmax(shell, kmax)          chemshell_set_kmax(shell, kmax)

CHEMshell *shell;                        integer       shell
int       *kmax;                         integer       kmax
```

Given a valid CHEMshell object, shell, **CHEMshell_set_kmax** will set the ending atomic orbital basis function. Kmax is constrained to be greater than or equal to 0.

CHEMshell_get_gauss

```
C                                        Fortran
integer                                  integer
CHEMshell_get_gauss(shell, gauss)        chemshell_get_gauss(shell, gauss)

CHEMshell *shell;                        integer       shell
CHEMgauss **gauss;                       integer       gauss
```

Given a valid CHEMshell object, shell, **CHEMshell_get_gauss** will return the root of the gaussian list for this shell.

CHEMshell_set_gauss

```
C                                        Fortran
integer                                  integer
CHEMshell_set_gauss(shell, gauss)        chemshell_set_gauss(shell, gauss)

CHEMshell *shell;                        integer       shell
CHEMgauss *gauss;                        integer       gauss
```

Given a valid CHEMshell object, shell, **CHEMshell_set_gauss** will assign the root of the gaussian list for this shell.

Gaussian Interface Functions

The CHEMgauss object represents the primitive functions for the shell. It is instanced by a CHEMshell object.

```
double         expo;        gaussian exponent
CHEMcoef      *coef;        contraction coefficient
double     coord[3];        cartesian coords of each gaussian
```

CHEMgauss_alloc

```
C                                   Fortran
CHEMgauss*                          integer
CHEMgauss_alloc()                   chemgauss_alloc()
```

The **CHEMgauss_alloc** function returns a CHEMgauss object to the caller. All data is initialized to 0, with all ancillary objects terminated. These Fortran and C functions will not return a CHEM error code; rather they will return 0 or NULL respectively.

CHEMgauss_free

```
C                                   Fortran
void                                integer
CHEMgauss_free(gauss)               chemgauss_free(gauss)

CHEMgauss  *gauss;                  integer gauss
```

Given a valid CHEMgauss object, gauss, **CHEMgauss_free** will traverse the list designated by gauss and free all ancillary CHEMcoef lists.

CHEMgauss_get

```
C                                   Fortran
int                                 integer
CHEMgauss_get(gauss, expo,          chemgauss_get(gauss, expo,
   coef, x, y, z)                     coef, x, y, z)

CHEMgauss    *gauss;                integer      gauss
CHEMcoef     **coef;                integer      coef
double       *expo;                 real*8       expo
double       *x;                    real*8       x
double       *y;                    real*8       y
double       *z;                    real*8       z
```

Given a valid CHEMgauss object gauss, **CHEMgauss_get** will return the components of gauss.

CHEMgauss_set

```
C                                       Fortran
int                                     integer
CHEMgauss_set(gauss, expo,              chemgauss_set(gauss, expo,
   coef, x, y, z)                         coef, x, y, z)

CHEMgauss    *gauss;                    integer      gauss
CHEMcoef     *coef;                     integer      coef
double       expo;                      real*8       expo
double       x;                         real*8       x
double       y;                         real*8       y
double       z;                         real*8       z
```

Given a valid CHEMgauss object gauss, **CHEMgauss_set** will assign the components of CHEMgauss object, gauss.

CHEMgauss_get_xyz

```
C                                       Fortran
int                                     integer
CHEMgauss_get_xyz(gauss, x, y, z)       chemgauss_get_xyz(gauss, x, y, z)

CHEMgauss    *gauss;                    integer      gauss
double       *x;                        real*8       x
double       *y;                        real*8       y
double       *z;                        real*8       z
```

Given a valid CHEMgauss object gauss, **CHEMgauss_get_xyz** will return the xyz coordinate of the CHEMgauss object, gauss.

CHEMgauss_set_xyz

```
C                                       Fortran
int                                     integer
CHEMgauss_set_xyz(gauss, x, y, z)       chemgauss_set_xyz(gauss, x, y, z)

CHEMgauss    *gauss;                    integer      gauss
double       x;                         real*8       x
double       y;                         real*8       y
double       z;                         real*8       z
```

Given a valid CHEMgauss object gauss, **CHEMgauss_set_xyz** will assign the xyz location of the CHEMgauss object, gauss.

CHEMgauss_get_expo

```
C                                       Fortran
int                                     integer
CHEMgauss_get_expo(gauss, expo)         chemgauss_get_expo(gauss, expo)

CHEMgauss     *gauss;                   integer      gauss
double        *expo;                    real*8       expo
```

Given a valid CHEMgauss object gauss, **CHEMgauss_get_expo** will assign the exponent of the CHEMgauss object, gauss.

CHEMgauss_set_expo

```
C                                       Fortran
int                                     integer
CHEMgauss_set_expo(gauss, expo)         chemgauss_set_expo(gauss, expo)

CHEMgauss     *gauss;                   integer      gauss
double        expo;                     real*8       expo
```

Given a valid CHEMgauss object gauss, **CHEMgauss_set_expo** will assign the exponent of the CHEMgauss object, gauss.

CHEMgauss_get_coef

```
C                                       Fortran
int                                     integer
CHEMgauss_get_coef(gauss, coef)         chemgauss_get_coef(gauss, coef)

CHEMgauss     *gauss;                   integer     gauss
CHEMcoef      **coef;                   integer     coef
```

Given a valid CHEMgauss object gauss, **CHEMgauss_get_coef** will return the root of the coefficient list, CHEMcoef, coef.

CHEMgauss_set_coef

```
C                                       Fortran
int                                     integer
CHEMgauss_set_coef(gauss, coef)         chemgauss_set_coef(gauss, coef)
```

```
CHEMgauss     *gauss;                  integer     gauss
CHEMcoef      *coef;                   integer     coef
```

Given a valid CHEMgauss object gauss, **CHEMgauss_set_coef** will assign the root of the coefficient list, CHEMcoef, coef.

Coefficient Interface Functions

The CHEMcoef object represents a list of coefficients for each CHEMgauss object. It is referenced by a CHEMgauss object.

```
double           val;   value
```

CHEMcoef_alloc

```
C                                     Fortran
CHEMcoef*                             integer
CHEMcoef_alloc()                      chemcoef_alloc()
```

The **CHEMcoef_alloc** function returns a CHEMcoef object to the caller. All data is initialized to 0 and there are no ancillary objects. These Fortran and C functions will not return a CHEM error code; rather they will return 0 or NULL respectively.

CHEMcoef_free

```
C                                     Fortran
void
CHEMcoef_free(coef)                   chemcoef_free(coef)

CHEMcoef    *coef;                    integer coef
```

Given a valid CHEMcoef object, coef, **CHEMcoef_free** will traverse and free all members of the CHEMcoef list.

CHEMcoef_get_val

```
C                                     Fortran
int                                   integer
CHEMcoef_get_val(coef, val)           chemcoef_get_val(coef, val)

CHEMcoef     *coef;                   integer      coef
double          *val;                 real*8       val
```

Given a valid CHEMcoef object, coef, **CHEMcoef_get_val**, will return the coefficient value.

CHEMcoef_set_val

```
C                                       Fortran
int                                     integer
CHEMcoef_set_val(coef, val)             chemcoef_set_val(coef, val)

CHEMcoef    *coef;                      integer     coef
double      val;                        real*8      val
```

Given a valid CHEMcoef object, coef, and a double, val, **CHEMcoef_set_val** will assign the coefficient value.

General Utility Interface Functions

This section of functions deals with data transformation and conversion. The functions here are general utilities for use in application programming with the CDK.

CHEMgen_util_rgb_to_int

```
C                                       Fortran
int                                     integer
CHEMgen_util_rgb_to_int( color,         chemgen_util_rgb_to_int( color,
     red, green, blue )                     red, green, blue )

int         *color;                     integer    color
float       red;                        real       red
float       green;                      real       green
float       blue;                       real       blue
```

Given a color value represented by the arguments, red, green and blue, **CHEMgen_util_rgb_to_int** converts these values to the appropriate integer representation.

CHEMgen_util_input_molecule

```
Fortran
integer
chemgen_util_input_molecule( mol_input )
integer    mol_input
```

This function is required for all Fortran programs that input molecules. The programmer should set *molecule* = **chemgen_util_input_molecule(***mol_input)* where *mol_input* is passed in to the compute function and *molecule* is an integer that represents a CHEMmolecule object. Both the **Fortran Con_Write** and **Fortran Elesta** example programs illustate its use. This call is necessitated by the Fortran interface to the actual *libchem* library, which is written in C.

CHEMgen_util_int_to_rgb

```
C                                       Fortran
int                                     integer
CHEMgen_util_int_to_rgb( color,         chemgen_util_int_to_rgb( color,
     red, green, blue )                     red, green, blue )

int         color;                       integer    color
float       *red;                        real       red
float       *green;                      real       green
float       *blue;                       real       blue
```

Given a color value represented by the argument, color, **CHEMgen_util_int_-to_rgb** converts the integer representation of color to the appropriate floating point values red, green and blue.

CHEMgen_util_update_molecule

```
C                                       Fortran
int                                     integer
CHEMgen_util_update_molecule( mol,      chemgen_util_update_molecule( mol,
   mode, nums, num_qps, num_bnds )          mode, nums, num_qps, num_bnds )

CHEMmolecule  **mol;                    integer     mol
int           mode;                     integer     mode
int          nums[ NUMCHEMTYPES ];     integer     nums[ NUMCHEMTYPES ]
int           *num_qps;                 integer     num_qps
int           *num_bnds;                integer     num_bnds
```

Given a valid CHEMmolecule object mol, **CHEMgen_util_update_molecule** will traverse the list, mol, and according to the mode flag (**MULTI/SINGLE**) update any and all the reference counts contained in the list. The nums array holds the number of each type encounterered. The nums array, which array index corresponds to which CHEM"object", and **NUMCHEMTYPES** are defined in */usr/avs/include/chemistry/CHEM_lm.h.* Refer to that include file to see how to interpret the results. Num_qps returns the total number of doubles used to represent the qp_page structures. Num_bnds returns the total number of line segments used to represent the proper connectivity. This accounts for all valid **CHEM_BOND** types.

CHEMmolecule_bld_candb

```
C                                       Fortran
int**                                   integer
CHEMmolecule_bld_candb( mol, sym,       chemmolecule_bld_candb( mol, sym,
    mode, nbnds, nexbnds )                  mode, nbnds, nexbnds )

CHEMmolecule **mol;                     integer      mol
int          sym;                       integer      sym
int          mode;                      integer      mode
int          **nbnds;                   integer      nbnds
int          **nexbnds;                 integer      nexbnds
```

Given a valid CHEMmolecule object mol, **CHEMmolecule_bld_candb**, traverses the molecular structure and constructs connectivity tables. The argument sym, either **ASYMMETRIC** or **SYMMETRIC**, determines the type of connectivity table created (see figure below). The argument mode determines whether to process the molecule as a **SINGLE** or **MULTI**ple CHEMmolecule. The return nbnds is the total number of bonds encountered in the query. The return nexbnds is total number of line segments necessary to represent the connectivity for all bonds. The function returns a segment of memory that contains the expanded connectivity tables laid out sequentially in memory.

```
  +---+----------------------+-----------+------+
  ^   | originating atom     | bond atom | type |
  |   |     .                |           |      |
Nbnds |     .                |           |      |
  |   |     .                |           |      |
  v   |     ending atom      |           |      |
  +---+----------------------+-----------+------+
```

The symmetric representation should be viewed as pointing halfway to the bonded atom. The asymmetric representation, on the other hand, points the entire way to the bond atom and produces a list one half the size of the symmetric list. Note that errors in symmetric lists may be seen as partially completed bonds; this is not necessarily the case for **ASYMMETRIC** representations.

CHAPTER 6

CDK EXAMPLES

Examples

The following examples for the on-line modules **Read structure file**, **Write structure file** and **Monopole elesta** are written in both Fortran and C. The examples are on-line in */usr/avs/examples/chemistry* and are listed here for your convienience. The **Read** and **Write structure file** modules provide classical examples of using the CHEM"object" accessor and list manipulation functions. The **Monopole elesta** modules use these same types of CHEM"object" information to produce an AVS field that contains the electrostatic potential around that molecule.

Fortran Examples

Fortran Con_Read

```
C------------------------------------------------------------------------
C      Module : CHEMcon_rf
C     Purpose : Read a structure file composed of a '.con' file which
C             : contains atom names, index, locations and connectivity and,
C             : if present, and associated '.fch' file, containing charges.
C             : CHEMcon_rf will create a single Molecule object from these
C             : files.
C            :
C   Data Types : The following data types are used
C              : CHEMmolecule
C              : CHEMatom
C              : CHEMcandb
C              : CDK Internal representation of CHEMatom user data.
C------------------------------------------------------------------------
C
C
C
C
C------------------------------------------------------------------------
C      avsinit_modules : defines the AVS modules and parameters
C------------------------------------------------------------------------
       subroutine avsinit_modules
        include 'avs/avs.inc'
```

```
      external con_r
      integer con_r,param
      integer dummy

C-------------------------------------------------------------------------
C       Define the module name and type.
C-------------------------------------------------------------------------
      call AVSset_module_name('Fortran con read','data')

C-------------------------------------------------------------------------
C       Create an output port that conatins the molecule
C-------------------------------------------------------------------------
      dummy = AVScreate_output_port('molecule','molecule')

C-------------------------------------------------------------------------
C       Define the file browser and connect it.
C-------------------------------------------------------------------------
      param=AVSadd_parameter('Structure files','string',' ',' ','.con')
      call AVSconnect_widget(param,'browser')

C-------------------------------------------------------------------------
C       Define the compute function that performs the work.
C-------------------------------------------------------------------------
      call AVSset_compute_proc(con_r)

      end

C-------------------------------------------------------------------------
C     AVS conpute - con_r
C
C     Function : con_r : read a structure file and associated formal
C                        charge file
C
C     Inputs   : filename
C     Outputs  : a CHEMmolecule
C-------------------------------------------------------------------------
       integer function con_r(mol_out,file_name)
       include 'avs/avs.inc'
       include 'cheminc/CHEMlong.inc'
      integer mol_out
       integer error
      integer TRUE,FALSE
       character*(*) file_name

      parameter (MAXATM=1000)
      parameter (MAXBONDS=6)
      parameter (TRUE=1)
      parameter (FALSE=0)

      double precision atom_x,atom_y,atom_z
      real atom_charge
       integer atom_index,atom_type,atom_con
      character*2 atom_label
      integer nat
      integer single_bond
```

```
        character*69 title

        include 'io_block.cmn'

        integer pos_dot,readcon,readfch,indx,is_fc,bond_indx
        integer do_color
        real do_radius
        character*256 fname

        integer atom,atom_list
        integer mol
        integer cnb,cnb_list

        integer idummy

C       Initialize mol

        mol = 0
C-------------------------------------------------------------------------
C       Set the return value of con_r to FALSE, as there are many ways to
C       return an error, but only one to return the correct answer(s).
C-------------------------------------------------------------------------
        con_r=FALSE

C-------------------------------------------------------------------------
C       Check if there's a valid filename
C-------------------------------------------------------------------------

        fname = ' '
        do indx=1,len(file_name)
          fname(indx:indx)=file_name(indx:indx)
        enddo

        pos_dot=index(file_name,'.con')

C-------------------------------------------------------------------------
C       Check to insure there is a name associated with the .con file
C-------------------------------------------------------------------------
        if(pos_dot.eq.0)
     1    return

C-------------------------------------------------------------------------
C       Read the data files into the common block
C-------------------------------------------------------------------------
        if(readcon(fname).eq.0)
     1    return

C-------------------------------------------------------------------------
C       If here, the file's been read - look for the formal charge file
C-------------------------------------------------------------------------
        fname(pos_dot:pos_dot+3)='.fch'
        if(readfch(fname).eq.0) then
          is_fc=FALSE
        else
          is_fc=TRUE
          if( chematom_init_user_data() .ne. 0 ) return
```

```
       endif

       fname=' '

C-------------------------------------------------------------------------
C      Clean up from the call to this function if appropriate
C-------------------------------------------------------------------------
       if (mol.ne.0)  call chemmolecule_free(mol)

C-------------------------------------------------------------------------
C      Create the initial molecule and atom objects.
C-------------------------------------------------------------------------
       mol=chemmolecule_alloc()
       atom_list=chematom_alloc()
       atom=atom_list

C-------------------------------------------------------------------------
C      Loop to add the data of each atom into the current atom object,
C      and check for errors.
C-------------------------------------------------------------------------
       do indx=1,nat
         if(chematom_set_inumber(atom,atom_index(indx)) .ne. 0)
     1        return
         if(chematom_set_name(atom,atom_label(indx)) .ne. 0)
     1        return
         if(chematom_set_color(atom,do_color(indx)) .ne. 0)
     1        return
         if(chematom_set_radius(atom,do_radius(indx)) .ne. 0)
     1        return
         if(chematom_set_xyz(atom,atom_x(indx),atom_y(indx),
     1                             atom_z(indx)) .ne. 0)
     2        return

C-------------------------------------------------------------------------
C       If formal charge data is present, then create the extended atom
C       object, and set the charge information.
C-------------------------------------------------------------------------
         if(is_fc.eq.TRUE) then
           if(chematom_alloc_user_data(atom) .ne. 0)
     1          return
           if(chematom_set_charge(atom,atom_charge(indx)) .ne. 0)
     1         return
           if(chematom_set_parent(atom,1) .ne. 0)
     1         return
         endif

C-------------------------------------------------------------------------
C       Create the initial connectivity and bond (candb) object for
C       the current atom object.  Loop over the connectivity
C       table and add candb objects as appropriate.
C-------------------------------------------------------------------------
          cnb_list=chemcandb_alloc()
          cnb=cnb_list
          do bond_indx=1,MAXBONDS
            if(atom_con(indx,bond_indx).gt.0) then
              if(chemcandb_set(cnb,atom_con(indx,bond_indx), 1) .ne. 0)
```

```
     1           return

C-------------------------------------------------------------------------
C       Add the current candb object to the connectivity list.
C-------------------------------------------------------------------------
             if(chemcandb_add(cnb_list,cnb) .ne. 0)
     1           return

C-------------------------------------------------------------------------
C       Create a new candb object.
C-------------------------------------------------------------------------
             cnb=chemcandb_alloc()
           endif
         enddo

C-------------------------------------------------------------------------
C       Delete the leftover candb object
C-------------------------------------------------------------------------
         call chemcandb_free(cnb)

C-------------------------------------------------------------------------
C       Attach the connectivity list to the current atom object.
C-------------------------------------------------------------------------
         if(chematom_set_candb(atom,cnb_list) .ne. 0)
     1        return

C-------------------------------------------------------------------------
C       Add the current atom object to the list of atoms.
C-------------------------------------------------------------------------
         if(chematom_add(atom_list,atom) .ne. 0)
     1        return

C-------------------------------------------------------------------------
C       Create a new atom object
C-------------------------------------------------------------------------
         atom=chematom_alloc()
       enddo

C-------------------------------------------------------------------------
C       Delete the leftover atom object.
C-------------------------------------------------------------------------
       call chematom_free(atom)

C-------------------------------------------------------------------------
C      Set the molecule object data.
C      Add the the atom list, set the name and number of atoms.
C-------------------------------------------------------------------------
       if( chemmolecule_set_atom(mol,atom_list) .ne. 0)
     1      return
       if( chemmolecule_set_name(mol,title) .ne. 0)
     1      return
       if( chemmolecule_set_natom(mol,nat) .ne. 0)
     1      return


C-------------------------------------------------------------------------
```

```
C      Assign the molecule object to the ouput port and set the return state.
C-------------------------------------------------------------------------
       mol_out=mol
       con_r=TRUE
       return
       end

C-------------------------------------------------------------------------
C      Function : readcon : reads the atom data .con file.
C       Input    : filename
C-------------------------------------------------------------------------
       integer function readcon(fname)

        character*(*)fname

       parameter (MAXATM=1000)
       parameter (MAXBONDS=6)
       parameter (TRUE=1)
       parameter (FALSE=0)

       integer indx,indx2

       double precision atom_x,atom_y,atom_z
        real atom_charge
       integer atom_index,atom_type,atom_con
       character*2 atom_label
       integer nat
       character*69 title

       include 'io_block.cmn'

       character*1 char1,char2

C-------------------------------------------------------------------------
C      Open the file - on failure return error
C-------------------------------------------------------------------------
C      open(unit=10,name=fname,access='sequential',type='old',err=800)
       open(UNIT=10,FILE=fname,ACCESS='SEQUENTIAL',STATUS='OLD',ERR=800)

C-------------------------------------------------------------------------
C      Get the first line - title and number of atoms
C-------------------------------------------------------------------------
       read(10,900)nat,title
900    format(i3,a69)

C-------------------------------------------------------------------------
C      Read the remainder of the file - individual atom entries
C-------------------------------------------------------------------------
       do indx=1,nat
         do indx2=1,MAXBONDS
           atom_con(indx,indx2)=0
         enddo

         read(10,901)char1,char2,atom_index(indx),
     1               atom_x(indx),atom_y(indx),atom_z(indx),
     2               atom_type(indx),
```

```
     3               (atom_con(indx,indx2),indx2=1,MAXBONDS)

901      format(1x,a1,a1,i5,3f12.6,i5,6i5)

          if (char1.eq.' ') then
            atom_label(indx)(1:1)=char2
          else
            atom_label(indx)(1:1)=char1
            atom_label(indx)(2:2)=char2
          endif
        enddo

C-------------------------------------------------------------------------
C      Close the file and return.
C-------------------------------------------------------------------------
        close(unit=10)
        readcon=TRUE
        return

C-------------------------------------------------------------------------
C       Error address
C-------------------------------------------------------------------------
800     readcon=FALSE
        return
        end

C-------------------------------------------------------------------------
C       Function : readfch : read the formal charge file
C        Input    : filename
C-------------------------------------------------------------------------
        integer function readfch(fname)

        character*(*) fname

        parameter (MAXATM=1000)
        parameter (MAXBONDS=6)
        parameter (TRUE=1)
        parameter (FALSE=0)

        integer indx

        double precision atom_x,atom_y,atom_z
         real atom_charge
        integer atom_index,atom_type,atom_con
        character*2 atom_label
        integer nat
        character*69 title

        include 'io_block.cmn'

C-------------------------------------------------------------------------
C       Open the file and read the formal charge data
C-------------------------------------------------------------------------
C       open(unit=11,name=fname,access='sequential',type='old',err=800)
        open(UNIT=11,FILE=fname,ACCESS='SEQUENTIAL',STATUS='OLD',ERR=800)
```

```
C-------------------------------------------------------------------------
C       Loop over the formal charges and read them into the common block
C-------------------------------------------------------------------------
       do indx=1,nat
         read(11,901)atom_charge(indx)
901      format(5x,f10.6)
       enddo

C-------------------------------------------------------------------------
C       close the file and return
C-------------------------------------------------------------------------
       close(unit=11)
       readfch=TRUE
       return

C-------------------------------------------------------------------------
C      Error address
C-------------------------------------------------------------------------
800    readfch=FALSE
       return
       end

C-------------------------------------------------------------------------
C      Function : do_radius : return the atom radius based on atom type
C-------------------------------------------------------------------------
       real function do_radius(indx)

       parameter (MAXATM=1000)
       parameter (MAXBONDS=6)
       parameter (TRUE=1)
       parameter (FALSE=0)

       integer indx

       double precision atom_x,atom_y,atom_z
        real atom_charge
       integer atom_index,atom_type,atom_con
       character*2 atom_label
       integer nat
       character*69 title

       include 'io_block.cmn'

       real radii(30)

       data radii/1.65,1.5 ,1.5 ,1.5 ,1.25,1.35,1.35,1.5 ,1.35,1.35,
     1            1.40,1.80,1.95,2.15,1.85,1.85,1.85,1.85,2.00,0.0 ,
     2            1.25,1.5 ,1.25,1.25,1.75,0.0 ,0.0, 1.25,0.0 ,0.0/


       if(atom_type(indx).ge.1.and.
     1    atom_type(indx).le.30) then
         do_radius=radii(atom_type(indx))
       else
         do_radius=0.0
       endif
```

```
        return
        end


C-------------------------------------------------------------------------
C       Function : do_color : return the atom color based on atom type
C-------------------------------------------------------------------------
        integer function do_color(indx)
         include 'cheminc/CHEMlong.inc'

        parameter (MAXATM=1000)
        parameter (MAXBONDS=6)
        parameter (TRUE=1)
        parameter (FALSE=0)

        integer indx

        double precision atom_x,atom_y,atom_z
         real atom_charge
        integer atom_index,atom_type,atom_con
        character*2 atom_label
        integer nat
        character*69 title
        integer err
        integer color(30)
        integer GREEN,WHITE,RED,BLU_GR,BLACK,GREY,YELLO,MAGNT

        include 'io_block.cmn'

         err = 0

         err = chemgen_util_rgb_to_int( GREEN, 0.0, 1.0, 0.0 )
         err = chemgen_util_rgb_to_int( WHITE, 1.0, 1.0, 1.0 )
         err = chemgen_util_rgb_to_int( RED  , 1.0, 0.0, 0.0 )
         err = chemgen_util_rgb_to_int( BLU_GR,0.0, 1.0, 1.0 )
         err = chemgen_util_rgb_to_int( BLACK, 0.0, 0.0, 0.0 )
         err = chemgen_util_rgb_to_int( GREY , 0.7, 0.7, 0.7 )
         err = chemgen_util_rgb_to_int( YELLO, 1.0, 1.0, 0.0 )
         err = chemgen_util_rgb_to_int( MAGNT, 1.0, 0.0, 1.0 )

         color(1) = GREEN
         color(2) = GREEN
         color(3) = GREEN
         color(4) = GREEN
         color(5) = WHITE
         color(6) = RED
         color(7) = RED
         color(8) = BLU_GR
         color(9) = BLU_GR
         color(10) =BLU_GR
         color(11) =GREEN
         color(12) =GREEN
         color(13) =GREEN
         color(14) =GREEN
```

```
      color(15) =YELLO
      color(16) =YELLO
      color(17) =YELLO
      color(18) =YELLO
      color(19) =GREY
      color(20) =BLACK
      color(21) =WHITE
      color(22) =GREEN
      color(23) =WHITE
      color(24) =WHITE
      color(25) =MAGNT
      color(26) =BLACK
      color(27) =BLACK
      color(28) =WHITE
      color(29) =BLACK
      color(30) =BLACK

      if (err.ne.0) then
        do_color=0
      else
       if(atom_type(indx).ge.1.and.
     1       atom_type(indx).le.30) then
         do_color=color(atom_type(indx))
       else
         do_color=0
       endif
      endif

     return
     end
```

Fortran Con_Write

```
C-------------------------------------------------------------------------
C      Module : CHEMcon_wf
C     Purpose : Output a structure file composed of a '.con' file which
C             : contains atom names, index, locations and connectivity and,
C             : if present, and associated '.fch' file, containing charges.
C             : CHEMcon_wf accepts a single Molecule object.
C            :
C   Data Types : The following data types are used
C             : CHEMmolecule
C             : CHEMatom
C             : CHEMcandb
C             : CDK Internal representation of CHEMatom user data.
C-------------------------------------------------------------------------
C
C
C
C
C-------------------------------------------------------------------------
C      avsinit_modules : define the AVS module and parameters
C-------------------------------------------------------------------------
       subroutine avsinit_modules
        include 'avs/avs.inc'

       external con_w
       integer con_w,param
       integer dummy

C-------------------------------------------------------------------------
C       Define the module name and type.
C-------------------------------------------------------------------------
       call AVSset_module_name('Fortran con write','renderer')

C-------------------------------------------------------------------------
C       Create an input port that conatins the molecule
C-------------------------------------------------------------------------
       dummy = AVScreate_input_port('molecule','molecule',REQUIRED)

C-------------------------------------------------------------------------
C       Define the file browser and connect it.
C-------------------------------------------------------------------------
       param=AVSadd_parameter('Structure files','string',' ',' ',
     1        '.con')
       call AVSconnect_widget(param,'browser')

C-------------------------------------------------------------------------
C       Define the compute function that performs the work.
C-------------------------------------------------------------------------
       call AVSset_compute_proc(con_w)

       end

C-------------------------------------------------------------------------
C     AVS compute - con_w
```

```
C
C     Function: con_t : write a structure file and associated formal charge
C                       file.
C
C     Inputs  : a CHEMmolecule
C     Outputs : a structure file (and formal charge file if appropriate)
C-------------------------------------------------------------------------------
        integer function con_w(mol_in,file_name)
        include 'avs/avs.inc'
        include 'cheminc/CHEMlong.inc'
       integer mol_in
        integer error
       integer TRUE,FALSE
        character*(*) file_name
       integer get_type
       character*2 get_label

       parameter (MAXATM=1000)
       parameter (MAXBONDS=6)
       parameter (TRUE=1)
       parameter (FALSE=0)

       double precision atom_x,atom_y,atom_z
       real atom_charge
        integer atom_index,atom_type,atom_con
       character*2 atom_label
       integer nat
       integer single_bond
       character*256 title,at_title

       integer bond_list

       include 'io_block.cmn'

       integer pos_dot,indx,is_fc,bond_indx
       character*256 fname,fname2

       integer atom,atom_list
       integer molecule
       integer cnb,cnb_list

       integer dummy,dum1,dum2
       integer struc_dims(15)

       real*8 tx,ty,tz
       real tchg
       dimension nbonds(1)

C-------------------------------------------------------------------------------
C      There are many ways to return with an error, but only one that
C      returns TRUE...
C-------------------------------------------------------------------------------

       con_w=FALSE

C-------------------------------------------------------------------------------
```

```
C      Check if there's a valid filename
C-----------------------------------------------------------------------

        fname  = ' '
        fname2 = ' '

       do indx=1,len(file_name)
         fname (indx:indx)=file_name(indx:indx)
          fname2(indx:indx)=file_name(indx:indx)
       enddo

       pos_dot=index(file_name,'.con')

C-----------------------------------------------------------------------
C       Check to insure there is a name associated with the .con file
C-----------------------------------------------------------------------
       if(pos_dot.eq.0)
     1    return

C-----------------------------------------------------------------------
C       Make the filename of the formal charge file
C-----------------------------------------------------------------------
        fname2(pos_dot:pos_dot+3)='.fch'

C-----------------------------------------------------------------------
C       Check the input molecule and load counters
C-----------------------------------------------------------------------
        molecule = chemgen_util_input_molecule(mol_in)
        if (chemgen_util_update_molecule(molecule,0,struc_dims,
     1      dum1,dum2) .ne. 0) return

       if (chemmolecule_get_natom(molecule,nat) .ne. 0)
     1    return

       if (chemmolecule_get_name(molecule,title) .ne. 0)
     1    return

        if (chemmolecule_get_atom(molecule,atom_list) .ne. 0)
     1    return

C-----------------------------------------------------------------------
C      Loop to gather the atom-information contained in the molecule:
C
C      location
C      label
C      index
C      charge
C
C      From the label, make an initial guess at the type
C-----------------------------------------------------------------------
        i=1
        do while (i .le. nat)
                if (chematom_get_xyz(atom_list,tx,ty,tz) .ne. 0)
     1              return
                if (chematom_get_charge(atom_list,tchg) .ne. 0)
     1              return
```

```
                if (chematom_get_name(atom_list,at_title) .ne. 0)
     1             return
                if (chematom_get_inumber(atom_list,indx) .ne. 0)
     1             return

                 atom_x(i)=tx
                 atom_y(i)=ty
                 atom_z(i)=tz

                 atom_charge(i)=tchg

                atom_index(i)=indx

                atom_label(i)(1:2)=at_title(1:2)

                atom_type(i) = get_type(atom_label(i))
                atom_label(i) = get_label(atom_label(i))

                 do indx=1,6
                  atom_con(i,indx)=0
                enddo

                 atom_list = chematom_get_next(atom_list)
                 i=i+1
         enddo

C-------------------------------------------------------------------------
C      Load the asymmetric bond list
C-------------------------------------------------------------------------
       bond_list = CHEMmolecule_bld_candb(molecule,0,0,nbptr,dum1)
       call do_join(%val(nbptr),%val(bond_list))

C-------------------------------------------------------------------------
C      Retype the atoms in the internal structure, write the file and
C       exit.
C-------------------------------------------------------------------------
       call retype

       call writecon(fname,fname2)
       con_w=TRUE
       return
       end

C-------------------------------------------------------------------------
C      Function: do_join : Use the data provided by molecule_bld_candb
C                        : to connect the atoms in the appropriate
C                        : fashion (for retype)
C-------------------------------------------------------------------------
       subroutine do_join(nbonds,bond_list)

       integer nbonds,bond_list(1)

       integer disp

        disp = 1
```

```
        do i=1,nbonds
          do j=1,bond_list(disp+2)
            iat=bond_list(disp)+1
            jat=bond_list(disp+1)+1
            call join(iat,jat)
          enddo

           disp=disp+3
        enddo

        return
        end

C-----------------------------------------------------------------------
C       Function: join: Connect the specified atoms
C-----------------------------------------------------------------------
        subroutine join(iat,jat)

        parameter (MAXBONDS = 6)
        parameter (MAXATM = 1000)

        double precision atom_x,atom_y,atom_z
        real atom_charge
         integer atom_index,atom_type,atom_con
        character*2 atom_label
        character*256 title

        integer nat

        include 'io_block.cmn'

        loc_1=-1
        loc_2=-1

        do i=1,MAXBONDS
          if ((atom_con(iat,i).eq.0).and.(loc_1.eq.-1))
     1      loc_1=i

           if ((atom_con(jat,i).eq.0).and.(loc_2.eq.-1))
     1      loc_2=i
         enddo

        if ((loc_1.ne.-1).and.(loc_2.ne.-1)) then
          atom_con(iat,loc_1)=jat
          atom_con(jat,loc_2)=iat
        endif

        return
        end

C-----------------------------------------------------------------------
C       Function : writecon : writes the atom data .con file.
C-----------------------------------------------------------------------
        subroutine writecon(fname,fname2)

         character*(*) fname,fname2
```

```
        parameter (MAXATM=1000)
        parameter (MAXBONDS=6)
        parameter (TRUE=1)
        parameter (FALSE=0)

        integer indx,indx2

        double precision atom_x,atom_y,atom_z
         real atom_charge
        integer atom_index,atom_type,atom_con
        character*2 atom_label
        integer nat
        character*256 title

        include 'io_block.cmn'

         real tchg

C-------------------------------------------------------------------------
C       Open the file - on failure return error
C-------------------------------------------------------------------------
C       open(unit=10,name=fname,access='sequential',type='unknown',
C    1       err=800)
         open(UNIT=10,FILE=fname,ACCESS='SEQUENTIAL',STATUS='UNKNOWN',
     1       ERR=800)

C-------------------------------------------------------------------------
C       Write the first line - title and number of atoms
C-------------------------------------------------------------------------
        write(10,900)nat,title(:69)
900     format(i3,a69)

C-------------------------------------------------------------------------
C       Write the remainder of the file - individual atom entries
C-------------------------------------------------------------------------
        do indx=1,nat
          write(10,901)atom_label(indx),atom_index(indx),
     1                 atom_x(indx),atom_y(indx),atom_z(indx),
     2                 atom_type(indx),
     3                 (atom_con(indx,indx2),indx2=1,MAXBONDS)
901       format(1x,a2,i5,3f12.6,i5,6i5)
        enddo

C-------------------------------------------------------------------------
C      Close the structure file and write a formal charge file if needed
C-------------------------------------------------------------------------
        close(unit=10)

        do i=1,nat
          tchg=tchg+abs(atom_charge(i))
        enddo

        if (tchg.le.0.01)
     1    return
```

```
C       open(unit=11,name=fname2,access='sequential',type='unknown',
C     1       err=800)
         open(UNIT=11,FILE=fname2,ACCESS='SEQUENTIAL',STATUS='UNKNOWN',
     1       ERR=800)

        do i=1,nat
          write(11,902)i,atom_charge(i)
902       format(i5,f10.6)
        enddo
c
        close(unit=11)

        return

C-------------------------------------------------------------------------
C       Error address
C-------------------------------------------------------------------------
800     return
        end

C-------------------------------------------------------------------------
C       Function: retype : retype all atoms to reflect MM2 values
C-------------------------------------------------------------------------
C  Mapping table: from type by label to MM2 atom type
C
C       Atom codes:
C
C                By Label        MM2
C                --------        ---

C          1     Hydrogen        sp3 Carbon
C          2     Oxygen          sp2 Carbon
C          3     Nitrogen        Carbonyl Carbon
C          4     Carbon          sp Carbon
C          5     Phosphorus      Hydrogen
C          6     Sulfur          -O-
C          7     Fluorine        =O
C          8     Chlorine        sp3 Nitrogen
C          9     Bromine         sp2 Nitrogen
C         10     Iodine          sp Nitrogen
C         11                     Fluorine
C         12                     Chlorine
C         13                     Bromine
C         14                     Iodine
C         15                     -S-
C         16                     =S- (S+) or =S
C         17                     S - oxide
C         18                     S - fone
C         19     Silicon         Silicon
C         20                     lone-pair
C         21                     H (N,O) alcohol
C         22                     Cyclopropane C
C         23                     N(H) amine
C         24                     COO(H) carboxyl
C         25                     Phosphorus
C         26                     B trigonal
```

```
C        27                      B tetrahedral
C        28                      H vinyl alcohol
C        29
C        30                      catchall atom

        subroutine retype

        parameter (MAXATM = 1000)
        parameter (MAXBONDS = 6)

        double precision atom_x,atom_y,atom_z
        real atom_charge
         integer atom_index,atom_type,atom_con
        character*2 atom_label
        character*256 title

        integer nat

        include 'io_block.cmn'

        integer new_atom_type(MAXATM)

        do i=1,nat

             if(atom_type(i).eq.1) then
               if(atom_type(atom_con(i,1)).eq.2) then
                      new_atom_type(i)=21
               else if(atom_type(atom_con(i,1)).eq.3) then
                      new_atom_type(i)=23
               else
                      new_atom_type(i)=5
               endif

             else if(atom_type(i).eq.2) then
               if(atom_con(i,1).eq.atom_con(i,2)) then
                      new_atom_type(i)=7
               else
                      new_atom_type(i)=6
               endif

             else if(atom_type(i).eq.3) then
               if(atom_con(i,1).eq.atom_con(i,2).and.
     1             atom_con(i,1).eq.atom_con(i,3)) then
                      new_atom_type(i)=10
               else if((atom_con(i,1).eq.atom_con(i,2)).or.
     1                 (atom_con(i,1).eq.atom_con(i,3)).or.
     2                 (atom_con(i,2).eq.atom_con(i,3))) then
                      new_atom_type(i)=9
               else
                      new_atom_type(i)=8
               endif

             else if(atom_type(i).eq.4) then
               if((atom_con(i,1).eq.atom_con(i,2).and.
     1              atom_con(i,1).eq.atom_con(i,3)).or.
     2             (atom_con(i,1).eq.atom_con(i,3).and.
```

```
3              atom_con(i,1).eq.atom_con(i,4)).or.
4             (atom_con(i,2).eq.atom_con(i,3).and.
5              atom_con(i,2).eq.atom_con(i,4)).or.
6             (atom_con(i,1).eq.atom_con(i,3).and.
7              atom_con(i,1).eq.atom_con(i,4))) then
                 new_atom_type(i)=4
           else if((atom_con(i,1).eq.atom_con(i,2).and.
1                 (atom_type(atom_con(i,1)).eq.2.or.
z                   atom_type(atom_con(i,1)).eq.3)).or.
2                  (atom_con(i,1).eq.atom_con(i,3).and.
3                 (atom_type(atom_con(i,1)).eq.2.or.
y                   atom_type(atom_con(i,1)).eq.3)).or.
4                  (atom_con(i,1).eq.atom_con(i,4).and.
5                 (atom_type(atom_con(i,1)).eq.2.or.
x                   atom_type(atom_con(i,1)).eq.3)).or.
6                  (atom_con(i,2).eq.atom_con(i,3).and.
7                 (atom_type(atom_con(i,2)).eq.2.or.
w                   atom_type(atom_con(i,2)).eq.3)).or.
8                  (atom_con(i,2).eq.atom_con(i,4).and.
9                 (atom_type(atom_con(i,2)).eq.2.or.
v                   atom_type(atom_con(i,2)).eq.3)).or.
a                  (atom_con(i,3).eq.atom_con(i,4).and.
b                 (atom_type(atom_con(i,3)).eq.2.or.
u                   atom_type(atom_con(i,3)).eq.3))) then
                  new_atom_type(i)=3
           else if((atom_con(i,1).eq.atom_con(i,2)).or.
1                  (atom_con(i,1).eq.atom_con(i,3)).or.
2                  (atom_con(i,1).eq.atom_con(i,4)).or.
3                  (atom_con(i,2).eq.atom_con(i,3)).or.
4                  (atom_con(i,2).eq.atom_con(i,4)).or.
5                  (atom_con(i,3).eq.atom_con(i,4))) then
                 new_atom_type(i)=2
           else
                 new_atom_type(i)=1
           endif

         else if(atom_type(i).eq.5) then
           new_atom_type(i)=25

         else if(atom_type(i).eq.6) then
           if((atom_con(i,1).eq.atom_con(i,2)).or.
1              atom_con(i,3).ne.0) then
                 new_atom_type(i)=16
           else
                 new_atom_type(i)=15
           endif

         else if(atom_type(i).eq.7) then
           new_atom_type(i)=11

         else if(atom_type(i).eq.8) then
           new_atom_type(i)=12

         else if(atom_type(i).eq.9) then
           new_atom_type(i)=13
```

```
                else if(atom_type(i).eq.10) then
                  new_atom_type(i)=14

                else if(atom_type(i).eq.19) then
                  new_atom_type(i)=19

                else if(atom_type(i).eq.30) then
                  new_atom_type(i)=30

                endif

        enddo
c
c       Restore the atom types to their proper places
c
        do i=1,nat
                atom_type(i)=new_atom_type(i)
        enddo
c
c       Remove multiple bonds
c
        do i=1,nat
                call debond(i)
        enddo
C
C       Return to caller
c
        return
        end

C-------------------------------------------------------------------------
C       Function: debond: remove all multiply-bound connections
C-------------------------------------------------------------------------
        subroutine debond(i)
c
c       this subroutine deletes multiple bonds from a .con file entry.
c       Repair of the connectivity will be done via redo.
c
        parameter (MAXATM = 1000)
        parameter (MAXBONDS = 6)

        double precision atom_x,atom_y,atom_z
        real atom_charge
         integer atom_index,atom_type,atom_con
        character*2 atom_label
        character*256 title

        integer nat

        include 'io_block.cmn'

        integer stack(15)!stack of bound atoms
c
c       Initialization
c
        nstack=0
```

```
c
       do j=1,6
              if(nstack.eq.0) then
                     nstack=nstack+1
                     stack(nstack)=atom_con(i,j)
                     goto 100   !"break"
              endif
c
c             If here, there is already entries on the stack
c
              iflag=0           !hit flag
              do k=1,nstack
                     if(atom_con(i,j).eq.stack(k)) iflag=1
              enddo
c
c             If there was a hit, delete the entry (otherwise put it
c             on the stack).
c
              if(iflag.eq.0) then
                     nstack=nstack+1
                     stack(nstack)=atom_con(i,j)
              else
                     atom_con(i,j)=0
              endif
c
100           continue
       enddo
c
c      Repair the entry
c
       call redo(i)
c
c      Go back
c
       return
       end

C-----------------------------------------------------------------------
C      Function: redo: remove all zero's from "within" the connections
C-----------------------------------------------------------------------
       subroutine redo(natom)
c
c      this subroutine removes zeros from a connectivity listing
c      (the trick is to use a piece of the bubble sort algorithm --
c      float the zeros to the end...)
c
       parameter (MAXATM = 1000)
       parameter (MAXBONDS = 6)

       double precision atom_x,atom_y,atom_z
       real atom_charge
        integer atom_index,atom_type,atom_con
       character*2 atom_label
       character*256 title

       integer nat
```

```
      include 'io_block.cmn'
c
c     sort the line
c
      do i=1,5
            do j=1,(6-i)
               if(atom_con(natom,j).lt.atom_con(natom,j+1)) then
                              ind=atom_con(natom,j)
                              atom_con(natom,j)=atom_con(natom,j+1)
                              atom_con(natom,j+1)=ind
               endif
            enddo
      enddo
c
c     go back
c
      return
      end

C-----------------------------------------------------------------------
C     Function: get_type : Attach an initial type based on atom label
C-----------------------------------------------------------------------
      integer function get_type(label)

      parameter (MAXTYPE = 30)

      character*2 label,label_array(MAXTYPE)
      integer type_array(MAXTYPE)

C     Note: these data arrays are HARD CODED

      data label_array/'H','O','N','C','P','S',
     1                  'F','CL','BR','I','SI',
     2                  19*'@@'/

      data type_array/ 1, 2, 3, 4, 5, 6, 7, 8, 9,10,
     1                19,-1,-1,-1,-1,-1,-1,-1,-1,-1,
     2                -1,-1,-1,-1,-1,-1,-1,-1,-1,-1/

      do i=1,MAXTYPE

        if (label_array(i)(2:2).eq.' ') then
          if(label(1:1).eq.label_array(i)(1:1)) then
            get_type=type_array(i)
            return
          endif
        else if (label.eq.label_array(i)) then
          get_type=type_array(i)
          return
        endif
      enddo

      get_type=-1

      return
```

```
      end

C-----------------------------------------------------------------------
C     Function: get_label : Attach the correct atom label
C-----------------------------------------------------------------------
      character*2 function get_label(label)

      parameter (MAXTYPE = 30)

      character*2 label,label_array(MAXTYPE)
      character*2 new_label(MAXTYPE)

C     Note: these data arrays are HARD CODED

      data label_array/'H','O','N','C','P','S',
     1                 'F','CL','BR','I','SI',
     2                 19*'@@'/

      data new_label/' H',' O',' N',' C',' P',' S',
     1               ' F','CL','BR',' I','SI',
     2                 19*'@@'/

      do i=1,MAXTYPE

        if (label_array(i)(2:2).eq.' ') then
          if(label(1:1).eq.label_array(i)(1:1)) then
            get_label=new_label(i)
            return
          endif
        else if (label.eq.label_array(i)) then
          get_label=new_label(i)
          return
        endif
      enddo

      get_label='@@'

      return
      end
```

Fortran Elesta

```
C-------------------------------------------------------------------------
C       Module : CHEMelestf
C      Purpose : Given a single molecule object create a AVS 3D scalar field
C              : containing the electrostatic potential of the molecule.
C              :
C              :                   ----    ----    ----    ----    q(i)
C              : AVSfield(xyz) =  \      \       \       \       ------
C              :                  /      /       /       /     r(x,y,z;i)
C              :                  ----    ----    ----    ----
C              :                  x       y       z       atom
C              :
C              :
C   Data Types : The following data types are used:
C              : CHEMmolecule
C              : CHEMatom
C              : CDK Internal representation of CHEMatom user data.
C-------------------------------------------------------------------------
C
C
C
C
C-------------------------------------------------------------------------
C
C       avsinit_modules : Register the appropriate AVS parameters
C                       :
C                       : Define the module name and type.
C                       : Create an input port that contains the molecule
C                       : Create an output port that contains the field
C                       : Define the Volume, grid and Dielectric resolutions.
C                       : Define the compute function that performs the work.
C-------------------------------------------------------------------------

       subroutine avsinit_modules
        include 'avs/avs.inc'

       external elesta
       integer elesta,param1,param2,param3
       integer param4
       integer err1,err2

C-------------------------------------------------------------------------
C   Define the module name and type.
C-------------------------------------------------------------------------
       call AVSset_module_flags(SINGLE_ARG_DATA)
       call AVSset_module_name('Fortran elesta','filter')

C-------------------------------------------------------------------------
C   Define the molecule input and field output
C-------------------------------------------------------------------------
       err1 = AVScreate_input_port('molecule','molecule',REQUIRED)
       err2 = AVScreate_output_port('field',
     1           'field 3D scalar 3-space rectilinear real')
```

```
C-------------------------------------------------------------------------
C   Define the Volume, grid and Dielectric resolutions.
C-------------------------------------------------------------------------
        param1=AVSadd_parameter('Volume expansion','integer', 3,2,10)
        call AVSconnect_widget(param1,'islider')

       param2=AVSadd_parameter('Grid resolution','integer', 10,2,100)
       call AVSconnect_widget(param2,'islider')

       param3=AVSadd_parameter('Dielectric constant','real', 1.0,1.0,
     1        100.0)
       call AVSconnect_widget(param3,'slider')

       param4=AVSadd_parameter('Distance-dependent dielectric',
     1      'boolean',0,0,1)
       call AVSadd_parameter_prop(param4,'width','integer',4)

C-------------------------------------------------------------------------
C   Define the compute function that performs the work.
C-------------------------------------------------------------------------
       call AVSset_compute_proc(elesta)
       end

C-------------------------------------------------------------------------
C     elesta  : The compute function called to create the 3D scalar field
C             :
C     input   : molecule : input molecule object
C             : volx     : volume expansion factor.
C             : res      : field resolution factor
C             : diec     : dielectric constant
C             : dist_dep : distance dependent dielectric constant.
C     output  : ep_field : AVS field.
C
C     Note: This module utilizes the monopole approximation for the
C           electrostatic potential.  Because of this, the value of the
C           potential is undefined within any atom sphere.
C-------------------------------------------------------------------------
       integer function elesta(mol_input,ep_field,volx,res,diec,
     1       dist_dep)
        include 'avs/avs.inc'
        include 'cheminc/CHEMlong.inc'

       integer mol_input,ep_field
       integer volx,res
       real diec
       integer dist_dep

       parameter (MAXATM=1000)
       parameter (MAXPTS=100000)
        character*256 mname
        character*128 errbuf

       real*8 tx,ty,tz
        real tchg,chgs(MAXATM)

       real atom_x(MAXATM),atom_y(MAXATM),atom_z(MAXATM)
```

```
        real point_x(MAXPTS),point_y(MAXPTS),point_z(MAXPTS)
        real xinc,yinc,zinc,xtmp,ytmp,ztmp
        real extents(6),tran(3)
        integer i,j,k,a,err
        real abs_min,abs_max

        integer atom_p
        integer struc_dims(15)

        real f_data(MAXPTS)
        integer fdim,res_array(3)
        integer data_p,points_p
         integer dum1
         integer dum2

         integer ocoords
         real coords
         dimension coords(1)

        integer natoms
        integer molecule

        double precision radius(MAXATM)
        real rad
        integer rad_flag

        natoms=0

C-------------------------------------------------------------------------
C       There are many ways to return an error - but only one way to
C       return true...
C-------------------------------------------------------------------------

        elesta = 0!return(FALSE)

C-------------------------------------------------------------------------
C       Obtain the name of the molecule object and update the object.
C        Note the all fortran functions expecting input molecules must
C        call the chemgen_util_input_molecule function.
C-------------------------------------------------------------------------
         molecule = chemgen_util_input_molecule(mol_input)
        if (chemmolecule_get_name(molecule,mname) .ne. 0)
     1     return
        if (chemgen_util_update_molecule(molecule,0,struc_dims,dum1,
     1      dum2) .ne. 0) return

C-------------------------------------------------------------------------
C       Obtain the number of atoms and prime the args for AVSdata_alloc
C-------------------------------------------------------------------------
        if (chemmolecule_get_natom(molecule,natoms) .ne. 0)
     1     return
        fdim=(res+1)**3
        res_array(1)=res+1
        res_array(2)=res+1
        res_array(3)=res+1
```

```
C-----------------------------------------------------------------------
C      Check against the defined paramer MAXPTS
C-----------------------------------------------------------------------
       if(fdim .gt. MAXPTS) then
         call AVSerror('Grid size exceeds storage')
         elesta=0
          return
        endif

C-----------------------------------------------------------------------
C      Create the AVS field
C-----------------------------------------------------------------------
       ep_field=AVSdata_alloc(
     1        'field 3D scalar 3-space rectilinear real', res_array)
      if (ep_field .eq. 0) then
         write(errbuf,90) field_descriptor
 90      format('Error allocating field ',A)
         call AVSerror(errbuf)
         return
      endif
C-----------------------------------------------------------------------
C      Loop over the molecule object's atoms and extract the location, charge
C       and radius;  first obtain the list of atoms.
C-----------------------------------------------------------------------
       if (chemmolecule_get_atom(molecule,atom_p) .ne. 0)
     1     return

       i=1
       do while (i .le. natoms)
              if (chematom_get_xyz(atom_p,tx,ty,tz) .ne. 0)
     1            return
              if (chematom_get_charge(atom_p,tchg) .ne. 0)
     1            return
              if (chematom_get_radius(atom_p,rad) .ne. 0)
     1            return

              atom_x(i)=tx
              atom_y(i)=ty
              atom_z(i)=tz

              radius(i)=rad

              chgs(i)=tchg

               atom_p = chematom_get_next(atom_p)
              i=i+1
       enddo

C-----------------------------------------------------------------------
C       Check if the formal charges deviate from zero - if not, exit
C-----------------------------------------------------------------------
        tchg=0.0
        do i=1,natoms
          tchg=tchg+abs(chgs(i))
        enddo
```

```
         if(tchg.eq.0.0) then
           call AVSwarning('No formal charges are present')
           elesta=0!return(FALSE)
           return
         endif

C-------------------------------------------------------------------------
C      Obtain the extents of the molecule object.
C-------------------------------------------------------------------------
       if (chemmolecule_extents(molecule,SINGLE,extents) .ne. 0)
     1     return

C-------------------------------------------------------------------------
C      Calculate the origin
C-------------------------------------------------------------------------
       do i=1,3
              tran(i)=(extents(i+3)-extents(i))/2.0+extents(i)
       enddo

C-------------------------------------------------------------------------
C      Find the absolute smallest and largest coordinate
C-------------------------------------------------------------------------
       abs_min= 100000.0
       abs_max=-100000.0

       abs_min=min(abs_min,extents(1))
       abs_min=min(abs_min,extents(2))
       abs_min=min(abs_min,extents(3))

       abs_max=max(abs_max,extents(4))
       abs_max=max(abs_max,extents(5))
       abs_max=max(abs_max,extents(6))

       do i=1,3
              extents(i)  =(abs_min*float(volx))+tran(i)
              extents(i+3)=(abs_max*float(volx))+tran(i)
       enddo

C-------------------------------------------------------------------------
C      Determine the field increments
C-------------------------------------------------------------------------
       xinc=(extents(4)-extents(1))/float(res)
       yinc=(extents(5)-extents(2))/float(res)
       zinc=(extents(6)-extents(3))/float(res)

C-------------------------------------------------------------------------
C      Determine the actual points
C-------------------------------------------------------------------------
       xtmp=0.0
       ytmp=0.0
       ztmp=0.0
       a=1

       do k=1,(res+1)
         do j=1,(res+1)
           do i=1,(res+1)
```

```
               point_x(a)=extents(1)+xtmp
               point_y(a)=extents(2)+ytmp
               point_z(a)=extents(3)+ztmp
               a=a+1
                xtmp=xtmp+xinc
             enddo

             ytmp=ytmp+yinc
             xtmp=0.0
           enddo

           ztmp=ztmp+zinc
           ytmp=0.0
         enddo

C-------------------------------------------------------------------------
C      Load the point into the field
C-------------------------------------------------------------------------
        points_p=AVSfield_points_offset(ep_field, coords, ocoords)
         call load_pnts(coords(ocoords+1),res,extents,xinc,yinc,zinc)

C-------------------------------------------------------------------------
C      Case one: no dielectric constant
C-------------------------------------------------------------------------
         if ((diec .le. 1.0) .and. (dist_dep .eq. 0)) then
           do a=1,fdim
             rad_flag=0
             tchg=0.0
             do i=1,natoms
                dist=sqrt((atom_x(i)-point_x(a))**2+
     1                    (atom_y(i)-point_y(a))**2+
     2                    (atom_z(i)-point_z(a))**2)

C-------------------------------------------------------------------------
C      Check if the point's within an atom sphere
C-------------------------------------------------------------------------
                if(dist .le. radius(i)) rad_flag=1

                tchg=tchg+chgs(i)/dist
             enddo

             if(rad_flag .ne. 1) then
               f_data(a)=tchg
             else
               f_data(a)=0.0
             endif
           enddo

C-------------------------------------------------------------------------
C      Case two: user-specified dielectric constant
C-------------------------------------------------------------------------
         else if ((diec .gt. 1.0) .and. (dist_dep .eq. 0)) then
           do a=1,fdim
             rad_flag=0
             tchg=0.0
             do i=1,natoms
```

```
               dist=sqrt((atom_x(i)-point_x(a))**2+
     1                   (atom_y(i)-point_y(a))**2+
     2                   (atom_z(i)-point_z(a))**2)

C-------------------------------------------------------------------------
C      Check if the point's within an atom sphere
C-------------------------------------------------------------------------
               if(dist .le. radius(i)) rad_flag=1

               tchg=tchg+chgs(i)/(dble(diec)*dist)
             enddo

             if(rad_flag .ne. 1) then
               f_data(a)=tchg
             else
               f_data(a)=0.0
             endif
           enddo

C-------------------------------------------------------------------------
C      Case three: distance-dependent dielectric
C-------------------------------------------------------------------------
       else if (dist_dep .eq. 1) then
         do a=1,fdim
           rad_flag=0
           tchg=0.0
           do i=1,natoms
               dist=sqrt((atom_x(i)-point_x(a))**2+
     1                   (atom_y(i)-point_y(a))**2+
     2                   (atom_z(i)-point_z(a))**2)

C-------------------------------------------------------------------------
C      Check if the point's within an atom sphere
C-------------------------------------------------------------------------

               if(dist .le. radius(i)) rad_flag=1

               tchg=tchg+chgs(i)/(dist*dist)
             enddo

             if(rad_flag .ne. 1) then
               f_data(a)=tchg
             else
               f_data(a)=0.0
             endif
           enddo
         endif

C-------------------------------------------------------------------------
C      Load the data into the field
C-------------------------------------------------------------------------
       data_p=AVSfield_data_ptr(ep_field)
       call load_data(%val(data_p),fdim,f_data)

C-------------------------------------------------------------------------
C      Exit the routine
```

```
C-------------------------------------------------------------------------
       elesta=1!return(TRUE)
       return
       end

C-------------------------------------------------------------------------
C      load_pnts : load the points into the field
C-------------------------------------------------------------------------
       subroutine load_pnts(points,res,extents,xinc,yinc,zinc)

       real points(1),extents(6)
       real xinc,yinc,zinc
       real tmp

       integer res,i

       tmp=0.0
       do i=1,(res+1)
              points(i)=extents(1)+tmp
              tmp=tmp+xinc
       enddo

       tmp=0.0
       do i=1,(res+1)
              points(i+(res+1))=extents(2)+tmp
              tmp=tmp+yinc
       enddo

       tmp=0.0
       do i=1,(res+1)
              points(i+(2*(res+1)))=extents(3)+tmp
              tmp=tmp+zinc
       enddo

       return
       end

C-------------------------------------------------------------------------
C     load_data : load the data into the field
C-------------------------------------------------------------------------
       subroutine load_data(data,fdim,f_data)

       real data(1),f_data(1)
       integer fdim,i

       do i=1,fdim
              data(i)=f_data(i)
       enddo

       return
       end
```

C Examples

C Con_Read and Con_Write

```
/*---------------------------------------------------------------*/
/* include files
/*---------------------------------------------------------------*/
#include <string.h>
#include <avs/avs.h>
#include <stdio.h>
#define CHEM_APPL
#include <CHEMmol.h>

/*---------------------------------------------------------------*/
/* local defines
/*---------------------------------------------------------------*/
#define TRUE  1
#define FALSE 0
#define ERROR -1
#define MAXTYPE 30
#define MAXBONDS 6
#define STACKSIZE 23
#define NMODS sizeof(module_list)/sizeof(module_list[0])

/*---------------------------------------------------------------*/
/* Local Data structures
/*---------------------------------------------------------------*/
/*     Structure file internal data structure:  */
/*                                               */
/*     label - 2 character atom label           */
/*     indexer - atom indexer                       */
/*     x,y,z - atom coordinates                 */
/*     type  - atom type (MM2 types)            */
/*     con   - atom connectivity (currently 6 bonds supported)  */
/*     charge- formal charge of atom (if present) */
/*---------------------------------------------------------------*/
typedef struct _atoms {
  char label[3];             /* label for this atom  */
  int indexer;                 /* indexer of this atom   */
  double x;                  /* x location     */
  double y;                  /* y location     */
  double z;                  /* z location     */
  int type;                  /* MM2 type       */
  int con[MAXBONDS];         /* connectivity   */
  float charge;              /* charge of the atom    */
} atoms;

/*---------------------------------------------------------------*/
/* static variables
/*---------------------------------------------------------------*/
static char *Title = NULL;   /* The name of the molecule */
static int Nat;              /* The nuymber of  atoms in this molecule*/
static atoms *A_list;        /* pointer to the head of the local sctructure */
```

```
/*-----------------------------------------------------------------*/
/* NB: the radii and color tables provide the mapping information */
/*      relating structure file atom type to appropriate chemical */
/*     usage.  Changing the order will produce misrepresentations*/
/*-----------------------------------------------------------------*/

/*-----------------------------------------------------------------*/
/* Radii Table
/*-----------------------------------------------------------------*/
/*
  Mapping table: from type by label to MM2 atom type

       Atom codes:

              By Label          MM2
              --------          ---

        1     Hydrogen          sp3 Carbon
        2     Oxygen            sp2 Carbon
        3     Nitrogen          Carbonyl Carbon
        4     Carbon            sp Carbon
        5     Phosphorus        Hydrogen
        6     Sulfur            -O-
        7     Fluorine          =O
        8     Chlorine          sp3 Nitrogen
        9     Bromine           sp2 Nitrogen
       10     Iodine            sp Nitrogen
       11                       Fluorine
       12                       Chlorine
       13                       Bromine
       14                       Iodine
       15                       -S-
       16                       =S- (S+) or =S
       17                       S - oxide
       18                       S - fone
       19     Silicon           Silicon
       20                       lone-pair
       21                       H (N,O) alcohol
       22                       Cyclopropane C
       23                       N(H) amine
       24                       COO(H) carboxyl
       25                       Phosphorus
       26                       B trigonal
       27                       B tetrahedral
       28                       H vinyl alcohol
       29
       30                       catchall atom
*/

static float radii[MAXTYPE] = {
  1.65,     /* Csp3    */
  1.5 ,     /* Csp2    */
  1.5 ,     /* Csp2=O  */
  1.5 ,     /* Csp     */
  1.25,     /* H       */
```

```
  1.35,     /* Osp3    */
  1.35,     /* Osp2    */
  1.5 ,     /* Nsp3    */
  1.35,     /* Nsp2    */
  1.35,     /* Nsp     */

  1.40,     /* F       */
  1.80,     /* CL      */
  1.95,     /* BR      */
  2.15,     /* I       */
  1.85,     /* S       */
  1.85,     /* S       */
  1.85,     /* S       */
  1.85,     /* S       */
  2.00,     /* SI      */
  0.0 ,

  1.25,     /* (O)H    */
  1.5 ,     /* Cycloprop C  */
  1.25,     /* (N)H    */
  1.25,     /* (COO)H  */
  1.75,     /* P       */
  0.0 ,
  0.0 ,
  1.25,     /* (HCC)H  */
  0.0 ,
  0.0
};

/*-----------------------------------------------------------------*/
/* Color Table
/*-----------------------------------------------------------------*/
static float  color[MAXTYPE][3] = {
  { 0.0, 1.0, 0.0 },   /* green   */
  { 0.0, 1.0, 0.0 },   /* green   */
  { 0.0, 1.0, 0.0 },   /* green   */
  { 0.0, 1.0, 0.0 },   /* green   */
  { 1.0, 1.0, 1.0 },   /* white   */
  { 1.0, 0.0, 0.0 },   /* red     */
  { 1.0, 0.0, 0.0 },   /* red     */
  { 0.0, 1.0, 1.0 },   /* bluegreen */
  { 0.0, 1.0, 1.0 },   /* bluegreen */
  { 0.0, 1.0, 1.0 },   /* bluegreen */

  { 0.0, 1.0, 0.0 },   /* green   */
  { 0.0, 1.0, 0.0 },   /* green   */
  { 0.0, 1.0, 0.0 },   /* green   */
  { 0.0, 1.0, 0.0 },   /* green   */
  { 1.0, 1.0, 0.0 },   /* yellow  */
  { 1.0, 1.0, 0.0 },   /* yellow  */
  { 1.0, 1.0, 0.0 },   /* yellow  */
  { 1.0, 1.0, 0.0 },   /* yellow  */
  { 0.7, 0.7, 0.7 },   /* grey    */
  { 0.0, 0.0, 0.0 },   /* black   */

  { 1.0, 1.0, 1.0 },   /* white   */
```

```
  { 0.0, 1.0, 0.0 },   /* green   */
  { 1.0, 1.0, 1.0 },   /* white   */
  { 1.0, 1.0, 1.0 },   /* white   */
  { 1.0, 0.0, 1.0 },   /* magenta */
  { 0.0, 0.0, 0.0 },   /* black   */
  { 0.0, 0.0, 0.0 },   /* black   */
  { 1.0, 1.0, 1.0 },   /* white   */
  { 0.0, 0.0, 0.0 },   /* black   */
  { 0.0, 0.0, 0.0 }    /* black   */
};

/*-----------------------------------------------------------------*/
/* AVS header - read_con
/*
/* Function : read_con : defines the AVS module and parameters
/*-----------------------------------------------------------------*/
int
read_con()
{
  int read_con_compute();
  char *getcwd();
  int param = 0;
  char dir[256];
  char *directory= NULL;

  /* Define the module output port and module name */
  AVScreate_output_port("molecule","molecule");
  AVSset_module_name("Read structure file",MODULE_DATA);

  /* Get the current working directory */
  directory = getcwd(NULL,256);
  sprintf(dir,"%s/",directory);

  /* Define what types of files we are searching for */
  param = AVSadd_parameter("Structure files","string",dir,NULL,".con");
  AVSconnect_widget(param,"browser");

  /* Define the AVS compute proc */
  AVSset_compute_proc(read_con_compute);
}

/*-----------------------------------------------------------------*/
/* AVS compute - read_con
/*
/* Function : read_con_compute : read a structure file and
/*                               associated formal charge file.
/* Inputs   : filename
/* Outputs  : a CHEMmolecule
/*-----------------------------------------------------------------*/
int
read_con_compute(output,filename)

     CHEMmolecule **output;
     char *filename;
{
  /* Define internal functions that we will call */
```

```
int readcon(),readfch();
unsigned long do_color();
float do_radius();

/* local stuff */
int is_fc = 0;
int indx = 0;
int bond_indx = 0;
char fname[256];
char *pos_dot = NULL;

/* Define the appropriate CHEM objects */
CHEMatom *atom = NULL;
CHEMatom *atom_head = NULL;
CHEMmolecule *mol = NULL;
CHEMcandb *cnb = NULL;
CHEMcandb *cnb_head = NULL;

/* Return w/o activity if there is no filename, or if the wrong kind of */
/* filename is specified */
if (filename == NULL)
  return(FALSE);

/* Preserve the filename */
strcpy(fname,filename);

/* Check if it's a structure file */
if ((pos_dot = strrchr(fname,'.')) == NULL)
  return(FALSE);

if (strcmp(pos_dot,".con"))
  return(FALSE);

/* Allocate space for the title */
if ((Title = (char*) malloc(80*sizeof(char))) == NULL) {
  AVSerror("Unable to malloc Title");
  return(FALSE);
}

/* Call the functions to read the structure file (and formal charge file if*/
/* one's present).  Set a flag if charges are present, and init UserData. */
if (readcon(fname))
  return(FALSE);

strcpy(pos_dot,".fch"); /* Change file suffix */
if (readfch(fname))
  is_fc = FALSE;
else {
  is_fc = TRUE;
  CHEMatom_init_user_data();
}

/* Make space for the first molecule and atom objects */
if((mol = (CHEMmolecule*)CHEMmolecule_alloc()) == NULL)
  AVSerror("Could not allocate molecule");
if((atom_head = atom = (CHEMatom*)CHEMatom_alloc()) == NULL)
```

```
  AVSerror("Could not allocate atom");

/* Loop to create the flushed atom entries */

for (indx = 0; indx < Nat; indx++) {
  if(CHEMatom_set_inumber(atom,A_list[indx].indexer)) AVSerror("set_inum");
  if(CHEMatom_set_name(atom,A_list[indx].label))    AVSerror("set name");
  if(CHEMatom_set_color(atom,(int) do_color(indx))) AVSerror("set_color");
  if(CHEMatom_set_radius(atom,do_radius(indx)))     AVSerror("set radius");
  if(CHEMatom_set_xyz(atom,A_list[indx].x,
                  A_list[indx].y, A_list[indx].z)) AVSerror("set_xyz");

  /* Allocate the internal user data structure to hold the charge */
  /* information, then set the charge (and parent molecule). */
  if (is_fc) {
    if(CHEMatom_alloc_user_data(atom))   AVSerror("alloc_user_data");
    if(CHEMatom_set_charge(atom,A_list[indx].charge))AVSerror("set_charge");
    if(CHEMatom_set_parent(atom,1))      AVSerror("set_parent");
  }

  /* For each atom create connectivity and bond information.  At this */
  /* point, all atoms are assumed to be connected by SINGLE bonds.    */
  if((cnb_head = cnb = (CHEMcandb*)CHEMcandb_alloc()) == NULL)
    AVSerror("Could not allocate connectiviry and bond data");

  for (bond_indx = 0; bond_indx < MAXBONDS; bond_indx++){
    if (A_list[indx].con[bond_indx]) {
     if(CHEMcandb_set(cnb,A_list[indx].con[bond_indx], CHEM_BOND_SINGLE))
       AVSerror("set connectivity and bond data");
     if(CHEMcandb_add(&cnb_head,cnb)) AVSerror("add cann and bond");
     if((cnb = (CHEMcandb*) CHEMcandb_alloc()) == NULL) AVSerror("candb");
    }
  }

  /* Free the leftover cnb and attach the list to the atom */
  CHEMcandb_free(cnb);
  if(CHEMatom_set_candb(atom,cnb_head)) AVSerror("set candb");
  if(CHEMatom_add(&atom_head,atom)) AVSerror("add atom");
  if((atom = (CHEMatom*) CHEMatom_alloc()) == NULL) AVSerror("alloc atom");
}

/* Free the leftover atom and attach the list to the molecule.  Set some */
/* other information in the CHEMmolecule. */
CHEMatom_free(atom);
if(CHEMmolecule_set_atom(mol,atom_head)) AVSerror( "set atom" );
if(CHEMmolecule_set_name(mol,Title))     AVSerror("set name");
if(CHEMmolecule_set_units(mol,ANGSTROMS))AVSerror("set_units");
if(CHEMmolecule_set_natom(mol,Nat))      AVSerror("set_natom");

/* Release local memory */
if (A_list) free(A_list);
if (Title)  free(Title);

/* Assign the output molecule and exit   */
*output = mol;
return(TRUE);
```

```
}

/*---------------------------------------------------------------------*/
/* Function : readcon : read a structure file and
/*                      and load into the internal atom structures
/* Inputs   : filename
/*---------------------------------------------------------------------*/
int
readcon(fname)
     char *fname;
{
  FILE *fopen();
  FILE *fp = NULL;
  char ioline[80];
  int indx = 0;
  int indx2 = 0;

  /*  Open the file for reading  */
  if ((fp = fopen(fname,"r")) == NULL)  return(ERROR);

  /* Get the first line - the #atoms and the title string  */
  fgets(ioline,73,fp);
  sscanf(ioline,"%3d",&Nat);

  /* locate the root of the filename */
  for (indx = 3; indx < strlen(ioline); indx++)
    if (ioline[indx] != '\n')
      Title[indx-3] = ioline[indx];
    else
      Title[indx-3] = '\0';

  /* Allocate space for the local structure   */
  if ((A_list = (atoms *) malloc(Nat*sizeof(atoms))) == NULL) {
    AVSerror("Unable to malloc A_list");
  }

  /* Read the individual atom entries  */
  for (indx = 0; indx < Nat; indx++) {
    for (indx2 = 0; indx2 < 6; indx2++)
      A_list[indx].con[indx2] = 0;

    fscanf(fp," %s %d %lf %lf %lf %d %d %d %d %d %d %d",
           A_list[indx].label,&A_list[indx].indexer,
          &A_list[indx].x,&A_list[indx].y,&A_list[indx].z,
          &A_list[indx].type,
          &A_list[indx].con[0],&A_list[indx].con[1],
          &A_list[indx].con[2],&A_list[indx].con[3],
          &A_list[indx].con[4],&A_list[indx].con[5]);
  }

  /* Close the file and return */
  fclose(fp);
  return(0);
}

/*-------------------------------------------------------------*/
```

```
/* Function : read_fch : read the formal charge file
/* Inputs   : filename
/*-------------------------------------------------------------*/
int
readfch(fname)

     char *fname;
{
  FILE *fopen();
  FILE *fp = NULL;

  int indx = 0;
  int dummy = 0;

  /* Open the file for reading */
  if ((fp = fopen(fname,"r")) == NULL) return(ERROR);

  /* Read the formal charges */
  for (indx = 0; indx < Nat; indx++)
    fscanf(fp," %d %f",&dummy,&A_list[indx].charge);

  /* Close the file and return */
  fclose(fp);
  return(0);
}

/*-------------------------------------------------------------------*/
/* Function:  do_radius - return the atom radius based on atom type
/*-------------------------------------------------------------------*/
float
do_radius(indx)

     int indx;
{
  if ((A_list[indx].type >= 1) && (A_list[indx].type <= 30))
    return(radii[A_list[indx].type-1]);
  else
    return(0.0);
}

/*-------------------------------------------------------------*/
/* Function: do_color - set the atom color based on atom type
/*-------------------------------------------------------------*/
unsigned long
do_color(indx)

     int indx;
{
  unsigned long j = 0;

  if ((A_list[indx].type >= 1) && (A_list[indx].type <= 30)) {
      CHEMgen_util_rgb_to_int( &j,
               color[A_list[indx].type-1][0],
               color[A_list[indx].type-1][1],
               color[A_list[indx].type-1][2] );
     return(j);
```

```
   }
   else
     return(0);
}

/*--------------------------------------------------------------------*/
/* AVS header - write_con
/*
/* Function : write_con : defines the AVS module and parameters
/*--------------------------------------------------------------------*/
int
write_con()
{
  int write_con_compute();
  char *getcwd();
  int param = 0;
  char *directory = NULL;
  char dir[256];

  /* Define the module input port and module name */
  AVScreate_input_port("molecule","molecule",REQUIRED);
  AVSset_module_name("Write structure file",MODULE_RENDER);

  /* Get the current directory */
  directory = getcwd(NULL,256);
  sprintf(dir,"%s\n",directory);

  /* Define what types of files we are searching for */
  param = AVSadd_parameter("Structure files","string",dir,NULL,".con");
  AVSconnect_widget(param,"browser");

  /* Define the AVS compute proc */
  AVSset_compute_proc(write_con_compute);
}

/*--------------------------------------------------------------------*/
/* AVS compute - write_con
/*
/* Function : write_con_compute : write a structure file and
/*                                associated formal charge file.
/* Inputs   : a CHEMmolecule
/* Outputs  : a structure file (and formal charge file if appropriate)
/*--------------------------------------------------------------------*/
int
write_con_compute(input,filename)

     CHEMmolecule *input;
     char *filename;
{
  /* Define the appropriate CHEM objects - only ATOM this time */
  CHEMatom *atom = NULL;

  /* Define other internal functions that we will call */
  void retype(),join();
  int writecon();
  int get_type();
```

```
int indx = 0;
int indx2 = 0;
char *at_title = NULL;
int *nbonds = NULL;
int *nbx = NULL;
int **bonds, **bonds_head;
int struct_dims[NUMCHEMTYPES];
int dummy = 0;
int atom_index = 0;
int *atom_map = NULL;
char fname[256];
char fname2[256];
char *pos_dot = NULL;

/* Remove the suffix from the filename (exit if nothing specified).  If */
/* it's in the correct format, copy it to make the name of the formal */
/* charge file. */
if (filename == NULL)  return(FALSE);
strcpy(fname,filename);

if ((pos_dot = strrchr(fname,'.')) == NULL)  return(FALSE);
if (strcmp(pos_dot,".con"))                  return(FALSE);

strcpy(fname2,fname);
strcpy(pos_dot,".fch");

/* Open the molecule, and extract the atom and connectivity information. */
/* First, verify that the counters in CHEMmolecule are correct. */
if(CHEMgen_util_update_molecule(input,SINGLE,struct_dims,&dummy,&dummy))
  AVSerror("update molecule");
if(CHEMmolecule_get_natom(input,&Nat)) AVSerror("get natoms");
if(CHEMmolecule_get_atom(input,&atom)) AVSerror("get atom");
if(CHEMmolecule_get_name(input,&Title)) AVSerror("get name");

/* Allocate space for the local atom table and the mapping array.  The */
/* mapping array is required to insure that the output structure file */
/* utilizes monotonically increasing atom indices. */

if ((A_list = (atoms*) malloc(Nat*sizeof(atoms))) == NULL) {
  AVSerror("Unable to malloc A_list");
  return(FALSE);
}

if ((atom_map = (int*) malloc(Nat*sizeof(int))) == NULL) {
  AVSerror("Unable to malloc atom_map");
  return(FALSE);
}

for (indx = 0; atom; atom = CHEMatom_get_next(&atom), indx++) {
  if(CHEMatom_get_inumber(atom,&atom_index)) AVSerror("get inum");
  if(CHEMatom_get_xyz(atom,&A_list[indx].x,
                      &A_list[indx].y,
                      &A_list[indx].z))     AVSerror("atom get xyz");
  A_list[indx].charge = -200.0;
  if(CHEMatom_get_charge(atom,&A_list[indx].charge)) AVSerror("get chrg");
```

```
    if(CHEMatom_get_name(atom,&at_title)) AVSerror("get name");
    strncpy(A_list[indx].label,at_title,3);
    A_list[indx].label[2] = '\0';

    atom_map[indx] = atom_index;
    A_list[indx].indexer = indx+1;
    A_list[indx].type = get_type(A_list[indx].label);
    A_list[indx].con[0] = A_list[indx].con[1] = A_list[indx].con[2] = 0;
    A_list[indx].con[3] = A_list[indx].con[4] = A_list[indx].con[5] = 0;
  }

  /* Get an asymmetric bond list */
  if((bonds_head = bonds =
      CHEMmolecule_bld_candb(input,ASYMETRIC,SINGLE,&nbonds,&nbx)) == NULL )
    AVSerror("cannot build alternate candb list");

  /* Using the bond list, make the connections   */
  for (indx = 0; indx < nbonds[0]; indx++)
    for (indx2 = 0; indx2 < *(*bonds+2); indx2++) {
      join(*(*bonds),*(*bonds+1));
      *(bonds) += 3;
  }

  /* Retype the atoms based on the just-created bond info, then output  */
  retype();
  writecon(fname2,fname);

  /* Release the memory  */

  if (A_list)    free(A_list);
  if (atom_map)  free(atom_map);
  if (bonds_head)free(bonds_head);
  if (nbonds)    free(nbonds);
  if (nbx)       free(nbx);
  if (Title)     free(Title);
  if (at_title)   free(at_title);

  return(TRUE);
}

/*-------------------------------------------------------------------------*/
/* Function: writecon : Output the (new) structure file
/* Inputs: filename
/*-------------------------------------------------------------------------*/
int
writecon(fname,fname2)

     char *fname;
     char *fname2;
{
  FILE *fopen();
  FILE *fp = NULL;
  int indx = 0;
  char filename[256];
```

```
  strcpy(filename,fname);

  /* Open the file for writing */
  if ((fp = fopen(filename,"w")) == NULL) {
    AVSwarning("Unable to open output structure file");
    return FALSE;
  }

  /* Trim the title to 69 characters, and output the header line  */
  if (strlen(Title) >= 69) Title[69] = '\0';

  if (strlen(Title))
    fprintf(fp,"%3d%s\n",Nat,Title);
  else
    fprintf(fp,"%3d \n",Nat);

  /* Loop to write each atom entry in turn */
  for (indx = 0; indx < Nat; indx++)
    fprintf(fp, " %2s%5d%12.6lf%12.6lf%12.6lf%5d%5d%5d%5d%5d%5d%5d\n",
            A_list[indx].label, A_list[indx].indexer, A_list[indx].x,
            A_list[indx].y, A_list[indx].z, A_list[indx].type,
            A_list[indx].con[0], A_list[indx].con[1], A_list[indx].con[2],
            A_list[indx].con[3], A_list[indx].con[4], A_list[indx].con[5]);

  /* Close the file and return  */
  fclose(fp);

  /*If there are formal charges, write a formal charge file */

  if (A_list[0].charge != -200.0) {
    strcpy(filename,fname2);

    /* Open the file for writing */
    if ((fp = fopen(filename,"w")) == NULL) {
      AVSwarning("Unable to open output formal charge file");
      return FALSE;
    }

    /* Write each formal charge in turn */

    for (indx = 0; indx < Nat; indx++)
      fprintf(fp,"%5d%10.6f\n",A_list[indx].indexer,A_list[indx].charge);

    fclose(fp);
  }
}

/*---------------------------------------------------------------------------*/
/* Function: join : Join the specified atoms
/* Inputs: the atom pair to join
/*---------------------------------------------------------------------------*/
void
join(iat,jat)

     int iat;
     int jat;
```

```
{

  int indx = 0;
  int loc_1 = 0;
  int loc_2 = 0;

  /*  Make the bond in the first "open" space in the atom's connctvty lists */
  for (loc_1 = loc_2 = -1, indx = 0; indx < MAXBONDS; indx++) {
    if ((A_list[iat].con[indx] == 0) && (loc_1 == -1))
      loc_1 = indx;
    if ((A_list[jat].con[indx] == 0) && (loc_2 == -1))
      loc_2 = indx;
  }

  if ((loc_1 != -1) && (loc_2 != -1)) {
    A_list[iat].con[loc_1] = jat+1;
    A_list[jat].con[loc_2] = iat+1;
  }
}

/*---------------------------------------------------------------------*/
/* Function: redo : remove all zero's from "within" the connections
/*---------------------------------------------------------------------*/
void
redo(atom)

     int atom;
{
  int indx = 0;
  int indx2 = 0;
  int temp = 0;

  /* Remove zero's by sorting them to the end of the entry */
  for (indx = 0; indx < (MAXBONDS-2); indx++)
    for (indx2 = 0; indx2 < ((MAXBONDS-1)-indx); indx2++)
      if(A_list[atom].con[indx2] < A_list[atom].con[indx2+1]) {
        temp = A_list[atom].con[indx2];
        A_list[atom].con[indx2] = A_list[atom].con[indx2+1];
        A_list[atom].con[indx2+1] = temp;
      }
}

/*---------------------------------------------------------------------*/
/* Function: debond : remove all multiply-bound connections
/*---------------------------------------------------------------------*/
void
debond( atom )

     int atom;
{
  void redo();
  int indx = 0;
  int indx2 = 0;
  int stack[15];
  int num_stack = 1;
  int flag = 0;
```

```
  /* Push the first entry on the stack, and check all succeeding entries */
  /* against the stack.  If on stack, zero, otherwise push onto stack. */
  stack[0] = A_list[atom].con[0];

  for (indx = 1; indx < MAXBONDS; indx++) {
    for (flag = indx2 = 0; indx2 < num_stack; indx2++)
      if (stack[indx2] == A_list[atom].con[indx])
        flag = TRUE;

    if (flag)
      A_list[atom].con[indx] = 0;
    else {
      stack[num_stack] = A_list[atom].con[indx];
      num_stack++;
    }
  }

  /* Repair the connection table */
  redo( atom );
}

/*---------------------------------------------------------------------------*/
/* Function: retype : retype all atoms to reflect MM2 atom types
/*---------------------------------------------------------------------------*/
void
retype()
{
  int indx;
  int *new_type;

  /* Allocate working space to hold the new types  */
  if ((new_type = (int*) malloc(Nat*sizeof(int))) == NULL) {
    AVSerror("Unable to malloc new_type");
    return;
  }

  for (indx = 0; indx < Nat; indx++) {
    if (A_list[indx].type == 1)
      if (A_list[A_list[indx].con[0]-1].type == 2)
        new_type[indx] = 21;
      else if (A_list[A_list[indx].con[0]-1].type == 3)
        new_type[indx] = 23;
      else
        new_type[indx] = 5;

    else if (A_list[indx].type == 2)
      if (A_list[indx].con[0] == A_list[indx].con[1])
        new_type[indx] = 7;
      else
        new_type[indx] = 6;

    else if (A_list[indx].type == 3)
      if ((A_list[indx].con[0] == A_list[indx].con[1]) &&
          (A_list[indx].con[0] == A_list[indx].con[2]))
        new_type[indx] = 10;
```

```
  else if ((A_list[indx].con[0] == A_list[indx].con[1]) ||
           (A_list[indx].con[0] == A_list[indx].con[2]) ||
           (A_list[indx].con[1] == A_list[indx].con[2]))
    new_type[indx] = 9;
  else
    new_type[indx] = 8;

else if (A_list[indx].type == 4)
  if (((A_list[indx].con[0] == A_list[indx].con[1]) &&
       (A_list[indx].con[0] == A_list[indx].con[2])) ||
      ((A_list[indx].con[0] == A_list[indx].con[2]) &&
       (A_list[indx].con[0] == A_list[indx].con[3])) ||
      ((A_list[indx].con[1] == A_list[indx].con[2]) &&
       (A_list[indx].con[1] == A_list[indx].con[3])) ||
      ((A_list[indx].con[0] == A_list[indx].con[1]) &&
       (A_list[indx].con[0] == A_list[indx].con[3])))
    new_type[indx] = 4;

  else if (((A_list[indx].con[0] == A_list[indx].con[1]) &&
           ((A_list[A_list[indx].con[0]-1].type == 2) ||
            (A_list[A_list[indx].con[0]-1].type == 3))) ||
           ((A_list[indx].con[0] == A_list[indx].con[2]) &&
           ((A_list[A_list[indx].con[0]-1].type == 2) ||
            (A_list[A_list[indx].con[0]-1].type == 3))) ||
           ((A_list[indx].con[0] == A_list[indx].con[3]) &&
           ((A_list[A_list[indx].con[0]-1].type == 2) ||
            (A_list[A_list[indx].con[0]-1].type == 3))) ||
           ((A_list[indx].con[1] == A_list[indx].con[2]) &&
           ((A_list[A_list[indx].con[1]-1].type == 2) ||
            (A_list[A_list[indx].con[1]-1].type == 3))) ||
           ((A_list[indx].con[1] == A_list[indx].con[3]) &&
           ((A_list[A_list[indx].con[1]-1].type == 2) ||
            (A_list[A_list[indx].con[1]-1].type == 3))) ||
           ((A_list[indx].con[2] == A_list[indx].con[3]) &&
           ((A_list[A_list[indx].con[2]-1].type == 2) ||
            (A_list[A_list[indx].con[2]-1].type == 3))))
    new_type[indx] = 3;

  else if ((A_list[indx].con[0] == A_list[indx].con[1]) ||
           (A_list[indx].con[0] == A_list[indx].con[2]) ||
           (A_list[indx].con[0] == A_list[indx].con[3]) ||
           (A_list[indx].con[1] == A_list[indx].con[2]) ||
           (A_list[indx].con[1] == A_list[indx].con[3]) ||
           (A_list[indx].con[2] == A_list[indx].con[3]))
    new_type[indx] = 2;

  else
    new_type[indx] = 1;

else if (A_list[indx].type == 5)
  new_type[indx] = 25;

else if (A_list[indx].type == 6)
  if ((A_list[indx].con[0] == A_list[indx].con[1]) ||
      (A_list[indx].con[2] != 0))
```

```
        new_type[indx] = 16;
      else
        new_type[indx] = 15;

    else if (A_list[indx].type == 7)
      new_type[indx] = 11;

    else if (A_list[indx].type == 8)
      new_type[indx] = 12;

    else if (A_list[indx].type == 9)
      new_type[indx] = 13;

    else if (A_list[indx].type == 10)
      new_type[indx] = 14;

    else if (A_list[indx].type == 19)
      new_type[indx] = 19;

    else if (A_list[indx].type == 30)
      new_type[indx] = 30;
  }

  /*
    Restore the types to their proper place
  */

  for (indx = 0; indx < Nat; indx++)
    A_list[indx].type = new_type[indx];

  /*
    Remove multiple bonds
  */

  for (indx = 0; indx < Nat; indx++)
    debond(indx);

  /*
    Free working space
  */

  if (new_type)
    free(new_type);

}

/*--------------------------------------------------------------------------*/
/* Function: get_type : attach a simple type based on atom label
/* Note: these arrays are HARD-CODED
/*--------------------------------------------------------------------------*/
static struct _atom_data {
  char *label;
  int  type;
} Atom_data[MAXTYPE] ={
  "H",   1,
  "O",   2,
```

```
  "N",   3,
  "C",   4,
  "P",   5,
  "S",   6,
  "F",   7,
  "CL",  8,
  "BR",  9,
  "I",  10,
  "SI", 19,
  "  ", MAXTYPE,
  "  ", MAXTYPE,
  "  ", MAXTYPE,
  "  ", MAXTYPE,
  "  ", MAXTYPE,
  "  ", MAXTYPE,
  "  ", MAXTYPE,
  "  ", MAXTYPE,
  "  ", MAXTYPE,
  "  ", MAXTYPE,
  "  ", MAXTYPE,
  "  ", MAXTYPE,
  "  ", MAXTYPE,
  "  ", MAXTYPE,
  "  ", MAXTYPE,
  "  ", MAXTYPE,
  "  ", MAXTYPE,
  "  ", MAXTYPE,
  "  ", MAXTYPE
};

/*-----------------------------------------------------------------*/
/* get_type : return the indexer of the atom type from atom table.
/*-----------------------------------------------------------------*/
int
get_type( strng )
     char *strng;
{
  int indx = 0;

  for (indx = 0; indx < MAXTYPE; indx++)
    if (!strcmp(Atom_data[indx].label,strng))
      return(Atom_data[indx].type);

  return(MAXTYPE);
}

/*-----------------------------------------------------------------*/
/* AVS init - for both read_con and write_con
/*-----------------------------------------------------------------*/
int ((*module_list[])()) = {
  read_con,
  write_con
};

int
AVSinit_modules()
```

```
{
        AVSinit_from_module_list(module_list,NMODS);
}
```

C Elesta

```
/*--------------------------------------------------------------------
       Module : CHEMelesta
      Purpose : Given a single molecule object create a AVS 3D scalar field
              : containing the electrostatic potential of the molecule.
              :
              :                   ----    ----    ----    ----    q(i)
              : AVSfield(xyz) =   \      \       \       \       ------
              :                   /      /       /       /     r(x,y,z;i)
              :                   ----    ----    ----    ----
              :                   x       y       z       atom
              :
              :
   Data Types : The following data types are used:
              : CHEMmolecule
              : CHEMatom
              : CDK Internal representation of CHEMatom user data.
/*--------------------------------------------------------------------*/

/*--------------------------------------------------------------------*/
/* Include files
/*--------------------------------------------------------------------*/
#include <stdio.h>
#include <math.h>
#include <avs/avs.h>
#include <avs/field.h>
#define CHEM_APPL
#include <CHEMmol.h>

/*--------------------------------------------------------------------*/
/* Local defines
/*--------------------------------------------------------------------*/
#define TRUE 1
#define FALSE 0

#define VOLX    "Volume expansion"
#define RES     "Grid resolution"
#define DIEC    "Dielectric constant"

#define MIN(x,y) (x < y) ? x : y
#define MAX(x,y) (x > y) ? x : y

/*--------------------------------------------------------------------
       avsinit_modules : Register the appropriate AVS parameters
                       :
                       : Define the module name and type.
                       : Create an input port that contains the molecule
                       : Create an output port that contains the field
                       : Define the Volume, grid and Dielectric resolutions.
                       : Define the compute function that performs the work.
/*--------------------------------------------------------------------*/

/*--------------------------------------------------------------------*/
/* AVS init modules */
```

```
/*-------------------------------------------------------------------------*/
void
AVSinit_modules()
{
  int elesta();

  AVSmodule_from_desc(elesta);
}

/*-------------------------------------------------------------------------*/
/* AVS header
/*-------------------------------------------------------------------------*/
int
elesta()
{

  int elesta_compute(),param = 0;

  /* Define the module name and type  */
  AVSset_module_name("Monopole elesta",MODULE_FILTER);

  /* Define the molecule and field output  */
  AVScreate_input_port("molecule","molecule",REQUIRED);
  AVScreate_output_port("field","field 3D scalar 3-space rectilinear real");

  /* Define the Volume, grid and Dielectric resolutions  */
  AVSconnect_widget(AVSadd_parameter(VOLX,"integer",2,2,10),"islider");
  AVSconnect_widget(AVSadd_parameter(RES,"integer",10,2,100),"islider");

  AVSconnect_widget(AVSadd_float_parameter(DIEC,1.0,1.0,100.0),"slider");

  param = AVSadd_parameter("Distance-dependent dielectric","boolean",0,0,1);
  AVSadd_parameter_prop(param,"width","integer",4);

  /* Define the compute function that performs the work */
  AVSset_compute_proc(elesta_compute);
}


/*-------------------------------------------------------------------------*/
/* AVS compute

     elesta  : The compute function called to create the 3D scalar field
             :
     input   : molecule : input molecule object
             : volx     : volume expansion factor.
             : res      : field resolution factor
             : diec     : dielectric constant
             : dist_dep : distance dependent dielectric constant.
     output  : ep_field : AVS field.

     Note: This module utilizes the monopole approximation for the
           electrostatic potential.  Because of this, the value of the
           potential is undefined within any atom sphere.
/*-------------------------------------------------------------------------*/
```

```
int elesta_compute(input,output,volx,res,diec,dist_dep)
CHEMmolecule *input;
AVSfield_float **output;
int volx;
int res;
float *diec;
int dist_dep;
{

  double tx,ty,tz,dist;
  float tchg,*chgs = NULL;
  float *atom_x = NULL,*atom_y = NULL,*atom_z = NULL;
  float *point_x = NULL,*point_y = NULL,*point_z = NULL;
  float xinc,yinc,zinc,xtmp,ytmp,ztmp;
  float extents[6],tran[3];
  int fdim,res_array[3];
  int i,j,k,a,dummy;
  float abs_min,abs_max;

  CHEMatom *atom = NULL;
  int struct_dims[NUMCHEMTYPES];

  float *f_data = NULL;
  float *pnts = NULL;
  int natoms = 0;

  double *radius = NULL;
  float rad;
  int rad_flag;

  /* Obtain the number of atoms and prime the args for field allocation */

  if(CHEMgen_util_update_molecule(input,SINGLE,struct_dims,&dummy,&dummy))
    AVSerror("update molecule");
  natoms = struct_dims[ATOM];
  fdim = (res+1)*(res+1)*(res+1);
  res_array[0] = res_array[1] = res_array[2] = res+1;

  /* Create the AVS field */
  *output = (AVSfield_float*)
            AVSdata_alloc("field 3D scalar 3-space rectilinear real",
            res_array);
  f_data = (*output)->data;
  pnts = (*output)->points;

  /* Allocate memory for the work arrays */
  if ((chgs = (float*) malloc(natoms*sizeof(double))) == NULL) {
    AVSerror("Monopole elesta: Unable to malloc charge");
    return(FALSE);
  }
  if ((atom_x = (float*) malloc(natoms*sizeof(float))) == NULL) {
    AVSerror("Monopole elesta: Unable to malloc atom_x");
    return(FALSE);
  }
  if ((atom_y = (float*) malloc(natoms*sizeof(float))) == NULL) {
    AVSerror("Monopole elesta: Unable to malloc atom_y");
```

```
  return(FALSE);
}
if ((atom_z = (float*) malloc(natoms*sizeof(float))) == NULL) {
  AVSerror("Monopole elesta: Unable to malloc atom_z");
  return(FALSE);
}

if ((point_x = (float*) malloc(fdim*sizeof(float))) == NULL) {
  AVSerror("Monopole elesta: Unable to malloc point_x");
  return(FALSE);
}
if ((point_y = (float*) malloc(fdim*sizeof(float))) == NULL) {
  AVSerror("Monopole elesta: Unable to malloc point_y");
  return(FALSE);
}
if ((point_z = (float*) malloc(fdim*sizeof(float))) == NULL) {
  AVSerror("Monopole elesta: Unable to malloc point_z");
  return(FALSE);
}

if ((radius = (double*) malloc(natoms*sizeof(double))) == NULL) {
  AVSerror("Monopole elesta: Unable to malloc radius");
  return(FALSE);
}

/* Loop over the molecule object's atoms and extract the location, charge */
/* and radius; first obtain the list of atoms. */
if(CHEMmolecule_get_atom(input,&atom)) AVSerror("get_atom");

for (i = 0; atom; atom = CHEMatom_get_next(&atom), i++) {
  if(CHEMatom_get_xyz(atom,&tx,&ty,&tz)) AVSerror("get_xyz");
  if(CHEMatom_get_radius(atom,&rad)) AVSerror("get_radius");

  tchg = 0.0;
  if(CHEMatom_get_charge(atom,&tchg)) AVSerror("get_charge");

  atom_x[i] = (float) tx;
  atom_y[i] = (float) ty;
  atom_z[i] = (float) tz;
  chgs[i]   =         tchg;

  radius[i] = (double) rad;
}

/* Check if the formal charges deviate from zero - if not, exit */

for (tchg = 0.0, i = 0; i < natoms; i++)
  tchg += (float)fabs((double)chgs[i]);

if (tchg == 0.0) {
  AVSwarning("No formal charges are present");

  /* Free up space upon exit - signal abnormal termination  */

  if (chgs)      free(chgs);
  if (atom_x)    free(atom_x);
```

```
  if (atom_y)    free(atom_y);
  if (atom_z)    free(atom_z);

  if (point_x)   free(point_x);
  if (point_y)   free(point_y);
  if (point_z)   free(point_z);

  if (radius)    free(radius);

  return(FALSE);
}

/* Obtain the extents of the molecule object */
if(CHEMmolecule_extents(input,SINGLE,extents)) AVSerror("get_extents");

/* Calculate the origin */
for (i = 0; i < 3; i++)
  tran[i] = ((extents[i+3]-extents[i])/2.0) + extents[i];

/* Find the absolute smallest and largest coordinate */

abs_min =  100000.0;
abs_max = -100000.0;

abs_min = MIN(abs_min,extents[0]);
abs_min = MIN(abs_min,extents[1]);
abs_min = MIN(abs_min,extents[2]);

abs_max = MAX(abs_max,extents[3]);
abs_max = MAX(abs_max,extents[4]);
abs_max = MAX(abs_max,extents[5]);

for (i = 0; i < 3; i++) {
  extents[i]   = (abs_min * (float) volx) + tran[i];
  extents[i+3] = (abs_max * (float) volx) + tran[i];
}

/*  Determine the field increments */
xinc = (extents[3]-extents[0]) / (float) res;
yinc = (extents[4]-extents[1]) / (float) res;
zinc = (extents[5]-extents[2]) / (float) res;

/* Determine the actual points */

for (xtmp = ytmp = ztmp = 0.0, a = k = 0; k <= res; k++) {
  for (j = 0; j <= res; j++) {
    for (i = 0; i <= res; i++) {
      point_x[a] = extents[0] + xtmp;
      point_y[a] = extents[1] + ytmp;
      point_z[a] = extents[2] + ztmp;
      a++;

      xtmp += xinc;
    }

    ytmp += yinc;
```

```
    xtmp = 0.0;
  }

  ztmp += zinc;
  ytmp = 0.0;
}

/*   Load the points into the field */
for (xtmp = 0.0, i = 0; i < (res+1); i++, xtmp += xinc)
  pnts[i] = extents[0] + xtmp;

for (ytmp = 0.0, i = 0; i < (res+1); i++, ytmp += yinc)
  pnts[(res+1)+i] = extents[1] + ytmp;

for (ztmp = 0.0, i = 0; i < (res+1); i++, ztmp += zinc)
  pnts[(2*(res+1))+i] = extents[2] + ztmp;

/*  Case one - no dielectric constant */

if ((*diec <= 1.0) && (!dist_dep))
  for (tchg = 0.0, a = 0; a < fdim; a++, tchg=0.0) {
    for (i = rad_flag = 0; i < natoms; i++) {
      dist = sqrt((double)
              (((atom_x[i]-point_x[a])*(atom_x[i]-point_x[a]))+
               ((atom_y[i]-point_y[a])*(atom_y[i]-point_y[a]))+
               ((atom_z[i]-point_z[a])*(atom_z[i]-point_z[a]))));

     /* Check if the point's within an atom sphere*/

     if (dist <= radius[i])
       rad_flag = TRUE;

      tchg += (chgs[i]/(float)dist);
    }

    if (!rad_flag)
     f_data[a] = tchg;
    else
     f_data[a] = 0.0;
  }

/* Case two - user-specified dielectric constant */

else if ((*diec > 1.0) && (!dist_dep))
  for (tchg = 0.0, a = 0; a < fdim; a++, tchg=0.0) {
    for (i = rad_flag = 0; i < natoms; i++) {
      dist = sqrt((double)
              (((atom_x[i]-point_x[a])*(atom_x[i]-point_x[a]))+
               ((atom_y[i]-point_y[a])*(atom_y[i]-point_y[a]))+
               ((atom_z[i]-point_z[a])*(atom_z[i]-point_z[a]))));

     /*
       Check if the point's within an atom sphere
     */

     if (dist <= radius[i])
```

```
            rad_flag = TRUE;

          tchg += (chgs[i]/( *diec * (float)dist));
        }

        if (!rad_flag)
         f_data[a] = tchg;
        else
         f_data[a] = 0.0;
      }

    /* Case three - distance-dependent dielectric */

    else if (dist_dep)
      for (tchg = 0.0, a = 0; a < fdim; a++, tchg=0.0) {
        for (i = rad_flag = 0; i < natoms; i++) {
          dist = sqrt((double)
                  (((atom_x[i]-point_x[a])*(atom_x[i]-point_x[a]))+
                   ((atom_y[i]-point_y[a])*(atom_y[i]-point_y[a]))+
                   ((atom_z[i]-point_z[a])*(atom_z[i]-point_z[a]))));

         /* Check if the point's within an atom sphere*/
         if (dist <= radius[i])
           rad_flag = TRUE;

          tchg += (chgs[i]/(float)(dist * dist));
        }

        if (!rad_flag)
         f_data[a] = tchg;
        else
         f_data[a] = 0.0;
      }

    /* Free up space upon exit */

    if (chgs)    free(chgs);
    if (atom_x)  free(atom_x);
    if (atom_y)  free(atom_y);
    if (atom_z)  free(atom_z);

    if (point_x) free(point_x);
    if (point_y) free(point_y);
    if (point_z) free(point_z);

    if (radius)  free(radius);

    return(TRUE);
  }
```

CHAPTER 7

CDK MODULE MAN PAGES

Introduction

The module man pages included with the CDK have been created to illustrate assorted classes of AVS modules. In addition these components serve as the foundation of a test suite to validate the library and MDT communication mechanisms.

NAME

Colorize molecule – color individual atoms in a molecule based on atomic radius, formal charge or molecular weight

SUMMARY

Name Colorize molecule

Availability This module is in the */usr/avs/chem_lib* directory

Type filter

Inputs colormap
molecule

Outputs molecule

Parameters

Name	*Type*
color scheme	choice (default radii)

DESCRIPTION

The **Colorize molecule** module alters the color of individual atoms in a molecule based on atomic radius, formal charge, or atomic weight. The range of values, combined with a colormap determines the color spectrum. This module assumes that the input molecule will contain information regarding atomic radius and weight, along with formal charge. If the required data is missing, the atom colors of the output molecule are undefined.

INPUTS

Molecule A list of one (or more) chemical structures with the appropriate data created within AVS.

OUTPUTS

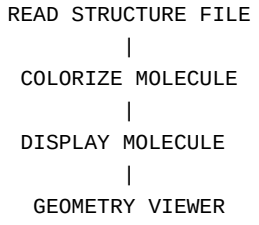
Molecule A list of one (or more) chemical structures (containing altered atom colors).

PARAMETERS

Color scheme

A set of radio buttons containing the following choices: **Radius** selects color based on atomic radius, **Formal Charge** selects color based on atomic (formal) charge and **Weight** selects color based on atomic weight.

EXAMPLE



RELATED MODULES

colorizer, contrast

LIMITATIONS

The input molecule must contain the data on which the color scheme is based.

Copy molecule

NAME

Copy molecule – generate two outputs from a single input molecule

SUMMARY

Name Copy molecule

Availability This module is in the */usr/avs/chem_lib* directory

Type filter

Inputs molecule

Outputs molecule copy #1
molecule copy #2

Parameters none

DESCRIPTION

This module copies the input molecule to two output molecules: one is the actual input molecule, and the other is its copy. This module is useful for shared memory implementations when multiple copies of a molecule are required.

INPUTS

Molecule A list of one (or more) chemical structures created within AVS.

OUTPUTS

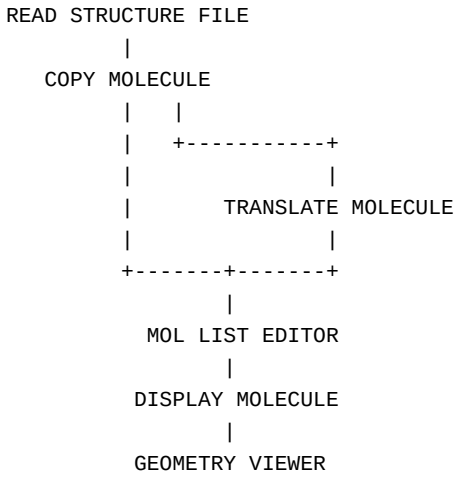
Molecule copy #1
The original molecule (input pointer passed to output pointer).

Molecule copy #2
The duplicate molecule.

PARAMETERS

None

EXAMPLE



RELATED MODULES

None

LIMITATIONS

None

NAME

Display molecule – display a molecule in various fashions

SUMMARY

Name Display molecule

Availability This module is in the */usr/avs/chem_lib* directory

Type mapper

Inputs molecule

Outputs geom

Parameters

Name	*Type*
operation	choice
atom scale	slider
transparency	slider
molecule selector	choice browser
bond highlight	choice

DESCRIPTION

This module allows the user to display the structure(s) contained in the input molecule in a variety of ways: colored sticks, ball and sticks, and solid spheres (of varying fractional radii). Particular bond types (single, double, etc.) can be highlighted for greater clarity. In addition, individual structures can be visualized with varying degrees of transparency, or hidden from view.

This module is important as it provides an example of complex memory management associated with operations involving the molecule data type. It also demonstrates how structural information can be transformed into AVS geometries, and how the Geometry Viewer can be manipulated from within a module.

INPUTS

Molecule A list of one (or more) chemical structures created within AVS.

OUTPUTS

Geom An AVS geom containing the structural representation.

PARAMETERS

Operation A radio button selector permitting the user to specify the mode of operation for **Display molecule**. The options are:

Variable Varies the fractional sphere radii. **Variable** is scaled by **Atom scale** below.

Ball+Stick displays structures as balls and sticks (default). Use **Atom** scale to change the radii.

Color Stick Displays structures as colored sticks. Use **Atom Scale** to change the radii.

Solid Displays structures as spheres. Use **Atom Scale** to change the radii.

Hide Allows the user to toggle the visibility of individual structures.

Atom scale A slider permitting the user to specify the fraction of the atomic radius to be used. The range is from 0 to 1. With **Color Stick** selected, the default is 0; with **Ball and Stick** it is .2; with **Solid** it is 1.

Transparency

A slider permitting the user to specify the amount of transparency to use when visualizing a structure. Note that the slider specifies the fractional "solidness" for the structure.

Molecule selector

A choice browser containing the title of each structure contained in the input molecule. Selecting an individual title applies the next operation selected to the appropriate structure.

Bond highlight

A radio button selector permitting the user to indicate that a particular bond type (**off, none, single, double, triple, hydrogen, aromatic**, or **disulfide**) should be highlighed in purple.

EXAMPLE

ucd to geom

LIMITATIONS

None

NAME

Fortran con read – read a structure file (and associated formal charge file) into AVS (in Fortran)

SUMMARY

Name Fortran con read

Availability This module is in the */usr/avs/examples/chemistry* directory

Type reader

Inputs none

Outputs molecule

Parameters

Name	*Type*
filename	file browser

DESCRIPTION

The Fortran con read module inputs a structure file. This file (suffix .con) contains the title and the number of atoms in the molecule, along with the location, type and connectivity of each atom in the molecule. If an associated formal charge file is present (suffix .fch), the formal charge for each atom is input. It produces a molecule that contains this information.

This module is an example of a reader that loads non-quantum chemical information relating to a molecule into AVS, and is functionally similar to readers associated with molecular mechanics and / or dynamics packages.

This module is written in Fortran to complement read structure file for pedagogic purposes. The use of Fortran results in inefficient use of memory, as all internal arrays must be dimensioned to hold the largest possible structure file.

INPUTS

None

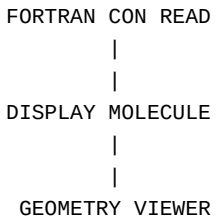
OUTPUTS

Molecule A single chemical structure containing geometric and connectivity information corresponding to that contained in the structure file.

PARAMETERS

Filename The particular structure file to use. The file suffix (.con) is assumed. The formal charge file need not be specified, as it will be used if available.

EXAMPLE



RELATED MODULES

read field, read volume, read image, read geom, read ucd

LIMITATIONS

Only one chemical structure can be read using Fortran con read. In addition, no Chemical Unit or Quantum information is contained in a structure file.

Fortran con write

NAME

Fortran con write – write a structure file (and associated formal charge file) from AVS (in Fortran)

SUMMARY

Name Fortran con write

Availability This module is in the */usr/avs/examples/chemistry* directory

Type data output

Inputs molecule

Outputs none

Parameters

Name	*Type*
filename	file browser

DESCRIPTION

The Fortran con write module outputs a structure file. This file (suffix *.con*) contains the title and the number of atoms in the molecule, along with the location, type and connectivity of each atom in the molecule. If the input molecule contains charge information, an associated formal charge file (suffix *.fch*) will be created. The format of these files is described in the *AVS Chemistry Developer's Guide*.

This module is an example of an output module that writes non-quantum chemical information relating to a molecule from within AVS, and it is functionally similar to output modules associated with molecular mechanics and / or dynamics packages.

This module is written in Fortran to complement write structure file for pedagogic purposes. The use of Fortran results in inefficient use of memory, as all internal arrays must be dimensioned to hold the largest possible structure file.

INPUTS

Molecule A list of one (or more) chemical structures created within AVS.

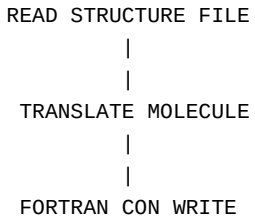
OUTPUTS

None

PARAMETERS

Filename The particular structure file to use. The file suffix (.con) is assumed. The formal charge file will be created if appropriate.

EXAMPLE



RELATED MODULES

Write structure file

LIMITATIONS

Only the first chemical structure from the input molecule list can be written at this time. In addition, no Chemical Unit or Quantum information is contained in a structure file.

NAME

Fortran elesta – calculate the molecular electrostatic potential, using the monopole (formal charge) approximation (in Fortran)

SUMMARY

Name Fortran elesta

Availability This module is in the */usr/avs/examples/chemistry* directory

Type filter

Inputs molecule

Outputs field 3D scalar 3-space rectilinear real

Parameters

Name	*Type*
volume expansion	integer
grid resolution	integer
dielectric constant	float
distance-dependent dielectric	toggle

DESCRIPTION

This module calculates the molecular electrostatic potential surrounding a molecule, using the monopole approximation. The potential at any point in space is given by:

$$\sum_{atoms} \frac{q(i)}{r*e}$$

where q(i) is the formal charge of atom i, r is the distance from the probe point to atom i, e is the dielectric constant and a +1 point charge is assumed. Using a distance-dependent dielectric constant replaces e with r.

The **Fortran elesta** module is an example of a filter that calculates a (classical) molecular property in a 3D field surrounding a molecule, and as such is the prototype of all such filters.

This module is written in Fortran to complement monopole elesta for pedagogic purposes. The use of Fortran provides both advantages and disadvantages: vector / parallel performance (where available) is improved somewhat, while the use of internally-dimensioned arrays places an upper-limit on the grid size that can be utilized.

INPUTS

Molecule A list of one (or more) chemical structures created within AVS.

OUTPUTS

Molecular electrostatic potential

A list of points, along with the value of the potential at each point.

PARAMETERS

Volume expansion

The multiplication factor applied to the molecular extents to determine the extents of the output field.

Grid resolution

The number of grid points dividing each axis of the output field.

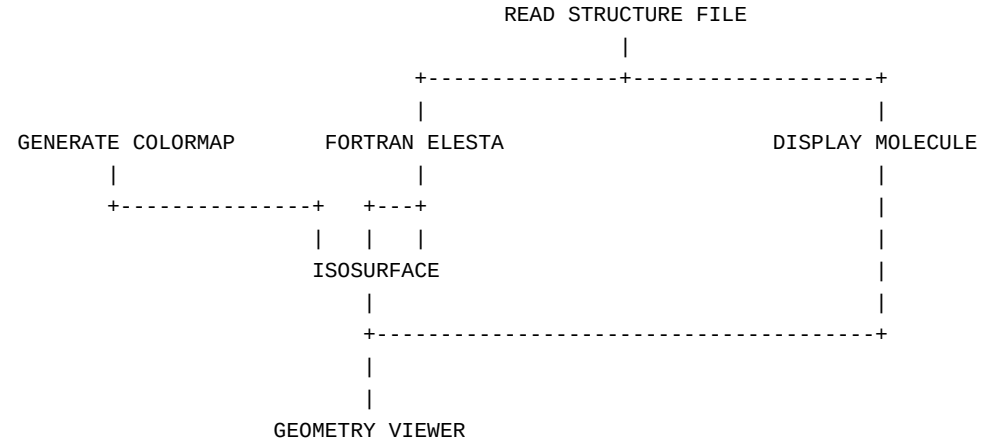
Dielectric constant

The value of the dielectric constant of the medium containing the molecule. Values greater than 1.0 model environmental effects on the electrostatic potential.

Distance-dependent dielectric

Indicate whether a distance-dependent dielectric constant should be used.

EXAMPLES



RELATED MODULES

Monopole elesta, compute gradient

LIMITATIONS

Only the first chemical structure from the input molecule list can be processed at this time. The molecular electrostatic potential is undefined at points within any atom using the monopole approximation, and is potentially inaccurate at points close to the Van der Waals surface.

Dimensions specified at compile time place a limit on the grid resolution that can be supported.

NAME

Mol list editor – edit a collection of molecular structures

SUMMARY

Name Mol list editor

Availability This module is in the */usr/avs/chem_lib* directory

Type filter

Inputs molecule1
molecule2

Outputs molecule

Parameters

Name	*Type*
operation	choice
mode	choice

DESCRIPTION

This module takes the input molecule1 and performs the list editing function specified by the current parameters. In its default state, **Mol list editor** will add molecule1 to molecule2—effectively joining the two molecules together.

There are two browsers that display the titles of structures contained in molecule 1 and molecule2. Molecule1's browser is titled **Molecule fragment titles**, and it is strictly informative. Molecule2's browser is titled **Molecule edit titles** and choosing a molecule from this browser will use it for the specified operation.

INPUTS

Molecule1
Molecule2 A list of one (or more) chemical structures created within AVS.

OUTPUTS

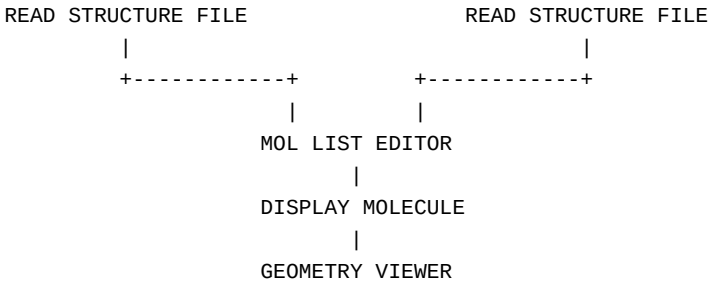
Molecule The suitably modified version of molecule2, reflecting the changes due the list manipulation operation.

PARAMETERS

Operation A radio button selector permitting the user to specify the mode of operation for **Mol list editor**. The options are: *None* ignores molecule1, and leaves molecule2 unchanged, *Add* adds molecule1 to molecule2, *Delete* deletes the specified structure from molecule2, *Replace* replaces the specified structure in molecule2 by molecule1, and *Insert* inserts molecule1 before the specified structure in molecule2.

Mode A radio button selector permitting the user to suspend the operation of **Mol list editor**. The options are: *Automatic* perform all list operations as selected and *Manual* suspends list operations.

EXAMPLE



RELATED MODULES

none

LIMITATIONS

Mol list editor can delete the last chemical structure in an input molecule list, and the result is a molecule list with no chemical structures. This NULL molecule list will not trigger down stream modules.

NAME

Molecular area – calculate the total surface area and the percent exposed surface area for individual atoms in a molecule

SUMMARY

Name Molecular area

Availability This module is in the */usr/avs/chem_lib* directory

Type filter

Inputs molecule

Outputs none

Parameters

Name	*Type*
number of arcs	integer
number of points	integer

DESCRIPTION

This module calculates the molecular surface area, along with the percent exposed surface area for each atom in a molecule using the following algorithm: (1) determine the surface area of each atom, (2) partition each atom into a number of concentric arcs, (3) subdivide each arc into the specified number of points, (4) determine the set of points not contained within another atom. This is the exposed surface area. These values are summed over all atoms to yield the molecular surface area.

The molecular surface area is displayed in a text string, while the percent exposed surface area for each atom is displayed in a text block browser.

INPUTS

Molecule A list of one (or more) chemical structures created within AVS.

OUTPUTS

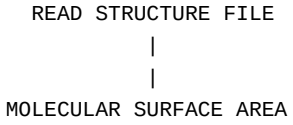
None

PARAMETERS

Number of arcs

The number of subdivisions used to partition each atom. This must be an odd number, so even entries will be incremented by one.

EXAMPLE



RELATED MODULES

Molecular volume

LIMITATIONS

Only the first chemical structure from the input molecule list can be processed at this time. The accuracy of the numerical integration algorithm is based on the number of arcs / points utilized.

Molecular volume

NAME

Molecular volume – calculate the molecular volume

SUMMARY

Name Molecular volume

Availability This module is in the */usr/avs/chem_lib* directory

Type filter

Inputs molecule

Outputs none

Parameters

Name	*Type*
number of points	integer

DESCRIPTION

This module calculates the molecular volume using the following algorithm: (1) determine the extents of the molecular geometry, and construct a box surrounding the molecule, (2) randomly place a specified number of points within the box, (3) determine the fraction of points which fall within any atom of the molecule. The molecular volume is this fraction multiplied by the box volume.

The molecular volume is displayed in a text string.

INPUTS

Molecule A list of one (or more) molecules.

OUTPUTS

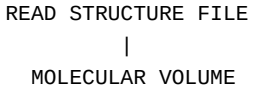
none

PARAMETERS

Number of points

The number of points to be used in the numerical integration algorithm.

EXAMPLE



RELATED MODULES

Molecular area

LIMITATIONS

Only the first chemical structure from the input molecule list can be processed at this time. The accuracy of the numerical integration algorithm is based on the number of points utilized.

NAME

Monopole elesta – calculate the molecular electrostatic potential, using the monopole (formal charge) approximation

SUMMARY

Name Monopole elesta

Availability This module is in the */usr/avs/chem_lib* directory

Type filter

Inputs molecule

Outputs field 3D scalar 3-space rectilinear real

Parameters

Name	*Type*
volume expansion	integer
grid resolution	integer
dielectric constant	float
distance-dependent dielectric	toggle

DESCRIPTION

This module calculates the molecular electrostatic potential surrounding a molecule, using the monopole approximation. The potential at any point in space is given by:

$$\sum_{atoms} \frac{q(i)}{r*e}$$

where q(i) is the formal charge of atom i, r is the distance from the probe point to atom i, e is the dielectric constant and a +1 point charge is assumed. Using a distance-dependent dielectric constant replaces e with r.

The **Monopole elesta** module is an example of a filter that calculates a (classical) molecular property in a 3D field surrounding a molecule, and as such is the prototype of all such filters.

INPUTS

Molecule A list of one (or more) chemical structures created within AVS.

OUTPUTS

Molecular electrostatic potential

A list of points, along with the value of the potential at each point.

PARAMETERS

Volume expansion

The multiplication factor applied to the molecular extents to determine the extents of the output field.

Grid resolution

The number of grid points dividing each axis of the output field.

Dielectric constant

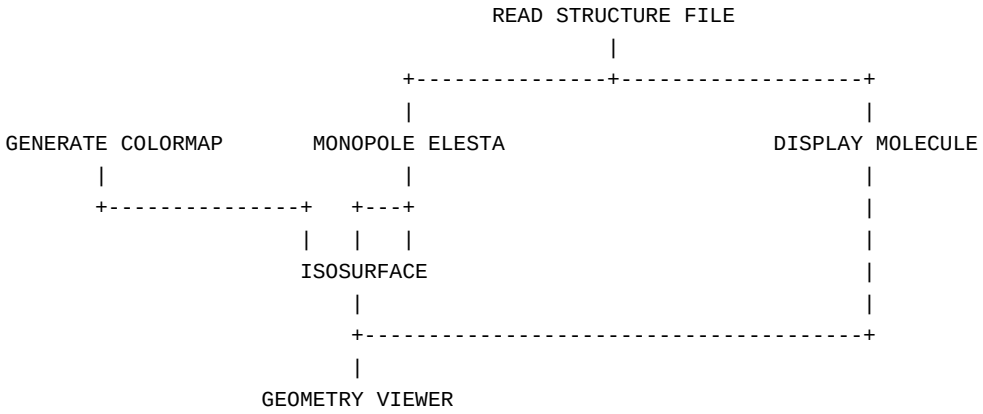
The value of the dielectric constant of the medium containing the molecule. Values greater than 1.0 model environmental effects on the electrostatic potential.

Distance-dependent dielectric

Indicate whether a distance-dependent dielectric constant should be used.

Monopole elesta

EXAMPLES



RELATED MODULES

compute gradient

LIMITATIONS

Only the first chemical structure from the input molecule list can be processed at this time. The molecular electrostatic potential is undefined at points within any atom using the monopole approximation, and is potentially inaccurate at points close to the Van der Waals surface.

NAME

Print molecule – print the contents of a molecule

SUMMARY

Name Print molecule

Availability This module is in the */usr/avs/chem_lib* directory

Type data output

Inputs molecule
Outputs none

Parameters	*Name*	*Type*
	temporary file	string typein

DESCRIPTION

The **Print molecule** module generates a hierarchical display of the entire contents of a molecule. A scratch file in */tmp* is used to hold the information, although the user can alter the filename if desired. A null entry results in the output being sent to stderr.

INPUTS

Molecule A list of one (or more) chemical structures created within AVS.

OUTPUTS

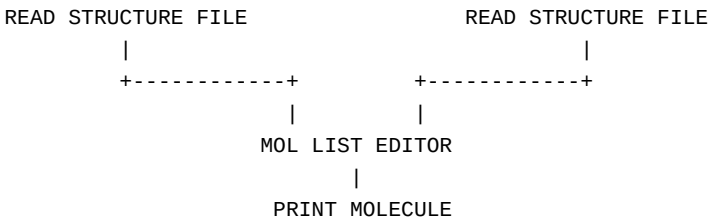
None

PARAMETERS

Temporary file

The scratch file containing the information generated by **Print molecule**.

EXAMPLE



RELATED MODULES

print field

LIMITATIONS

none

Read structure file

NAME

Read structure file – read a structure file (and associated formal charge file) into AVS

SUMMARY

Name Read structure file

Availability This module is in the */usr/avs/chem_lib* directory

Type reader

Inputs none

Outputs molecule

Parameters

Name	*Type*
filename	file browser

DESCRIPTION

The **Read structure file** module inputs a structure file. This file (suffix *.con*) contains the title and the number of atoms in the molecule, along with the location, type and connectivity of each atom in the molecule. If an associated formal charge file is present (suffix *.fch*), the formal charge for each atom is input. The format of these files is described in the *AVS Chemistry Developer's Guide*. It produces a molecule that contains this information.

This module is an example of a reader that loads non-quantum chemical information relating to a molecule into AVS, and is functionally similar to readers associated with molecular mechanics and / or dynamics packages.

INPUTS

None

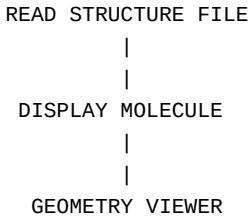
OUTPUTS

Molecule A single chemical structure containing geometric and connectivity information corresponding to that contained in the structure file.

PARAMETERS

Filename The particular structure file to use. The file suffix (.con) is assumed. The formal charge file need not be specified, as it will be used if available.

EXAMPLE



RELATED MODULES

read field, read volume, read image, read geom, read ucd

LIMITATIONS

Only one chemical structure can be read using Read structure file. In addition, no Chemical Unit or Quantum information is contained in a structure file.

NAME

Select molecule – select individual structures out of a molecule

SUMMARY

Name	Select molecule	
Availability	This module is in the */usr/avs/chem_lib* directory	
Type	filter	
Inputs	molecule	
Outputs	molecule	
Parameters	*Name*	*Type*
	choice	choice browser

DESCRIPTION

The **Select molecule** module permits a user to select individual structures from a molecule. A choice browser containing the titles of each structure is provided, and a molecule containing only the selected structure is output.

This module provides a mechanism for selecting individual entries from a list contained in the input molecule, thus acting as a gating mechanism for an AVS network. One potential use would be to permit individual structures in a reaction mechanism simulation (generated by a complex reader) to be processed through modules such as Monopole elesta.

INPUTS

Molecule A list of one (or more) chemical structures created within AVS.

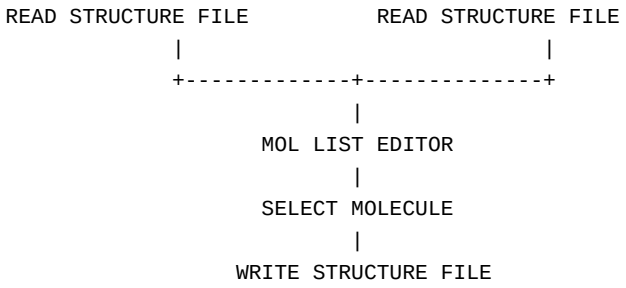
OUTPUTS

Molecule Contains the user-specified structure extracted from the input list.

PARAMETERS

Choice The title of each structure in the molecule.

EXAMPLE



RELATED MODULES

Mol list editor, extract scalar, extract vector

LIMITATIONS

The **Select molecule** module assumes that individual molecules in the input molecule list will have unique titles. If these are omitted or unclear, the overall usefulness of this module will be reduced.

Symmetric float

NAME

Symmetric float – send a positive and negative floating point number to one or more modules' floating point parameter port(s)

SUMMARY

Name Symmetric float

Availability This module is in the */usr/avs/chem_lib* directory

Type data

Inputs none

Outputs negative float
positive float

Parameters

Name	*Type*
value	float

DESCRIPTION

The **Symmetric float** module sends a positive and negative floating point number (same absolute value) to one or more floating point parameter ports on one or more AVS modules. It enables a user to simultaneously control positive and negative floating point parameter input using a single input widget.

Before you can connect **Symmetric float** to a receiving module, the parameter port must be made visible. To do this, call up the receiving module's Editor Window pane by pressing the middle or right mouse button on the module icon dimple. Next, scan the "Parameters" list to find the parameter to plug into. Position the cursor over that parameter's button and press any mouse button. When the Parameter Editor window appears, click any mouse button on its Port Visible switch. A purple parameter port should appear on the module icon. Connect this port to either the positive or negative port on the Symmetric float module in the usual manner.

INPUTS

None

OUTPUTS

Negative float

The negative floating point value is sent to all modules with floating point parameter ports connected to the **Symmetric float** module.

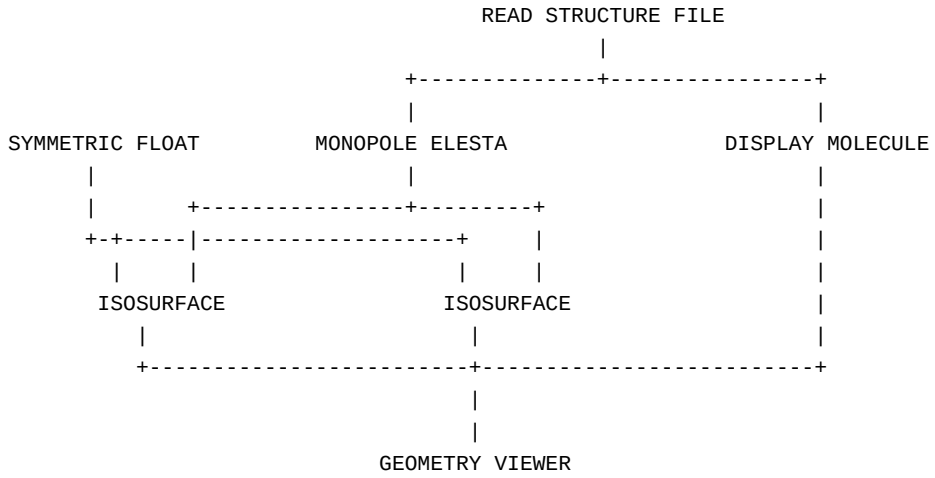
Positive float

The positive floating point value is sent to all modules with floating point parameter ports connected to the **Symmetric float** module.

PARAMETERS

Value The ±user-supplied floating point value to be sent to the module(s) floating point parameter port(s).

EXAMPLE



RELATED MODULES

float

LIMITATIONS

Care must be taken to ensure that the values sent by **Symmetric float** are within a range that is reasonable for the receiving module(s).

Translate molecule

NAME

Translate molecule – center a molecule at the center of charge, center of mass, center of coordinated or translate a molecule along the x, y or z-axis.

SUMMARY

Name Translate molecule

Availability This module is in the */usr/avs/chem_lib* directory

Type filter

Inputs molecule

Outputs molecule

Parameters

Name	*Type*
operation	choice
reset	toggle
xtran	float
ytran	float
ztran	float

DESCRIPTION

This molecule allows the user to center the structures contained in the input molecule at their center of mass, charge or coordinates or translate all of the structures x, y or z-axes. The translations are cumulative, and can be undone by selecting the Reset button.

If the input molecule lacks the required information to perform a selected operation, the user is informed. When centering is done, the displacement vector is provided in a text widget.

This module is important as it provides an example of operations on the molecule data type that avoid the use of intermediate arrays for all processing.

INPUTS

Molecule A list on one (or more chemical structures created within AVS.

OUTPUTS

Molecule A list on one (or more) chemical structures (suitable translated).

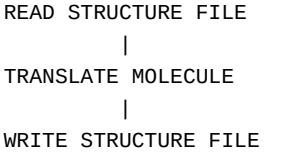
PARAMETERS

Operation A radio button selector permitting the user to specify the mode of operation for translate molecule. The options are: *Center coord* center by atomic coordinates, *Center charge* center by atomic (formal) charge, *Center weight* center by atomic weight (if present) and, *Translate* translate the molecule per user specification.

Reset Undo all specified translations.

Xtran
Ytran
Ztran The magnitude of the displacement along the (x,y,z)-axis to apply.

EXAMPLE



RELATED MODULES

none

LIMITATIONS

Simultaneous translations along multiple axes are not supported at this time.

Write structure file

NAME

Write structure file – write a structure file (and associated formal charge file) from AVS

SUMMARY

Name Write structure file

Availability This module is in the */usr/avs/chem_lib* directory

Type data output

Inputs molecule

Outputs none

Parameters

Name	*Type*
filename	file browser

DESCRIPTION

The Write structure file module outputs a structure file. This file (suffix *.con*) contains the title and the number of atoms in the molecule, along with the location, type and connectivity of each atom in the molecule. If the input molecule contains charge information, an associated formal charge file (suffix *.fch*) will be created. This format is documented in the *AVS Chemistry Developer's Guide*.

This module is an example of an output module that writes non-quantum chemical information relating to a molecule from within AVS, and is functionally similar to output modules associated with molecular mechanics and / or dynamics packages.

INPUTS

Molecule A list of one (or more) chemical structures created within AVS.

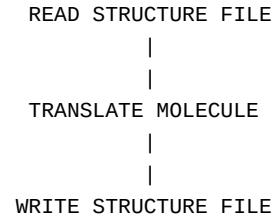
OUTPUTS

None

PARAMETERS

Filename The particular structure file to use. The file suffix (.con) is assumed. The formal charge file will be created if appropriate.

EXAMPLE



RELATED MODULES

read structure file

LIMITATIONS

Only the first chemical structure from the molecule input list can be written at this time. In addition, no Chemical Unit or Quantum information is contained in a structure file.