Cantera User's Guide

Fortran Version

Release 1.2

D. G. Goodwin

November, 2001

Division of Engineering and Applied Science
California Institute of Technology
Pasadena, CA
E-mail: dgoodwin@caltech.edu

CONTENTS

(draft November 29, 2001)

This document is the Fortran 90 version of the Cantera tutorial. If you are interested in using Cantera from C++ or Python, versions of this document exist for those languages as well.

CHAPTER

ONE

Getting Started

1.1 What is Cantera?

Cantera is a collection of object-oriented software tools for problems involving chemical kinetics, thermodynamics, and transport processes. Among other things, it can be used to conduct kinetics simulations with large reaction mechanisms, to compute chemical equilibrium, to evaluate thermodynamic and transport properties of mixtures, to evaluate species chemical production rates, to conduct reaction path analysis, to create process simulators using networks of stirred reactors, and to model non-ideal fluids. Cantera is still in development, and more capabilities continue to be added.

1.2 Typical Applications

Cantera can be used in many different ways. Here are a few.

Use it in your own reacting-flow codes. Cantera can be used in Fortran or C++ reacting-flow simulation codes to evaluate properties and chemical source terms that appear in the governing equations. Cantera places no limits on the size of a reaction mechanism, or on the number of mechanisms that you can work with at one time; it can compute transport properties using a full multicomponent formulation; and uses fast, efficient numerical algorithms. It even lets you switch reaction mechanisms or transport property models dynamically during a simulation, to adaptively switch between inexpensive/approximate models and expensive/accurate ones based on local flow conditions.

It is well-suited for numerical models of laminar flames, flow reactors, chemical vapor deposition reactors, fuel cells, engines, combustors, etc. Any existing code that spends most of its time evaluating kinetic rates (a common situation when large reaction mechanisms are used) may run substantially faster if ported to Cantera. (Cantera's kinetics algorithm, in particular, runs anywhere from two to four times faster, depending on the platform, than that used in some other widely-used packages.)

Use it for exploratory calculations. Sometimes you just need a quick answer to a simple question, for example:

- if air is heated to 3000 K suddenly, how much NO is produced in 1 sec?
- What is the adiabatic flame temperature of a stoichiometric acetylene/air flame?
- What are the principal reaction paths in silane pyrolysis?

With Cantera, answering any of these requires only writing a few lines of code. If you are comfortable with Fortran or C++ you can use either of these, or you can write a short Python script, which has the advantage that it can be run immediately without compilation. Python can also be used interactively. Or you can use one of the stand-alone applications that come with Cantera, which requires no programming at all, other than writing an input file.

Use it in teaching. Cantera is ideal for use in teaching courses in combustion, reaction engineering, transport processes, kinetics, or similar areas. Every student can have his or her own copy and use it from whatever language or application he or she prefers. There are no issues of cost, site licenses, license managers, etc., as there are for most commercial packages. For this reason, Cantera-based applications are also a good choice for software that accompanies textbooks in these fields.

Run simulations with your own kinetics, transport, or thermodynamics models. Cantera is designed to be customized and extended. You are not locked in to using a particular equation of state, or reaction rate expressions, or anything else. If you need a different kinetics model than one provided, you can write your own and link it in. The same goes for transport models, equations of state, etc. [Currently this requires C++ programming, but in an upcoming release this will be possible to do from Fortran.]

Make reaction path diagrams and movies. One of the best ways to obtain insight into the most important chemical pathways in a complex reaction mechanism is to make a reaction path diagram, showing the fluxes of a conserved element through the species due to chemistry. But producing these by hand is a slow, tedious process. Cantera can automatically generate reaction path diagrams at any time during a simulation. It is even possible to create reaction path diagram movies, showing how the chemical pathways change with time, as reactants are depleted and products are formed.

Create stirred reactor networks. Cantera implements a general-purpose stirred reactor model that can be linked to other ones in a network. Reactors can have any number of inlets and outlets, can have a time-dependent volume, and can be connected through devices that regulate the flow rate between them in various ways. Closed-loop controllers may be installed to regulate pressure, temperature, or other properties.

Using these basic components, you can build models of a wide variety of systems, ranging in complexity from simple constant-pressure or constant-volume reactors to complete engine simulators.

Simulate multiphase pure fluids. Cantera implements several accurate equations of state for pure fluids, allowing you to compute properties in the liquid, vapor, mixed liquid/vapor, and supercritical states. (This will soon be expanded to include multiphase mixture models.)

These capabilities are only the beginning. Some new features scheduled to be implemented in an upcoming release are non-ideal and multiphase reacting mixtures, surface and interfacial chemistry, sensitivity analysis, models for simple reacting flows (laminar flames, boundary layers, flow in channels, etc.), and electrochemistry. Other capabilities, or interfaces to other applications, may also be added, depending on available time, support, etc. If you have specific things you want to see Cantera support, you are encouraged to become a Cantera developer, or support Cantera development in other ways to make it possible. See http://www.cantera.org/support for more information.

1.3 Installing Cantera

If you've read this far, maybe you're thinking you'd like to try Cantera out. All you need to do is download a version for your platform from http://www.cantera.org and install it.

1.3.1 Windows

If you want to install Cantera on a PC running Windows, download and run 'cantera12.msi'. (If for some reason this doesn't work, you can download 'cantera12.zip' instead.)

If you only intend to run the stand-alone applications that come with Cantera, then you don't need a compiler. The executable programs are in folder 'bin' within the main Cantera folder.

If you intend to use Cantera in your own Fortran applications, you need to have Compaq (formerly Digital) Visual Fortran 6.0 or later. If you do, look in the 'win32' folder within the main Cantera folder. There you will find project and workspace files for Cantera.

To load the Cantera Fortran workspace into Visual Studio, open file 'Cantera_F90.dsw'.

1.3.2 unix/linux

If you use unix or linux, you will need to download the source code and build Cantera yourself, unless the web site contains a binary version for your platform. To build everything, you will need both a C++ compiler and a Fortran 90 compiler. A configuration script is used to set parameters for Makefiles. In most cases, the script correctly configures Cantera without problem.

1.3.3 The Macintosh

Cantera has not been ported to the Macintosh platform, but there is no reason it cannot be. If you successfully port it to the Macintosh, please let us know and we will add the necessary files to the standard distribution.

> Note that the Cantera Fortran interface requires Fortran 90, so if you are using the free `g77` compiler, you will need to upgrade.

1.3.4 Environment Variables

Before using Cantera, environment variable CANTERA_ROOT should be set to the top-level Cantera directory. If you are using Cantera on a PC running Windows, you can alternatively set WIN_CANTERA_ROOT. (This is defined so that users of unix-like environments that run under Windows (e.g. CygWin) can define CANTERA_ROOT in unix-like format (`/usr/local/cantera-1.2`) and WIN_CANTERA_ROOT in DOS-like format (`C:\CANTERA-1.2`).

In addition, you can optionally set CANTERA_DATA_DIR to a directory where input files should be looked for, if not found in the local directory. By default, Cantera will look in 'CANTERA_ROOT/data'.

1.4 Running Stand-Alone Applications

Cantera is distributed with a few simple application programs. These are located in the 'apps' subdirectory. For example, one such program is `zero`, a general-purpose zero-dimensional kinetics simulator. All of these applications include the source code, unix Makefiles, and Visual Studio project files, so they can also function as starting points to build your own applications.

1.5 The Cantera Fortran 90 Interface

Cantera consists of a *kernel* written in C++, and a set of language interface libraries that allow using Cantera from other programming languages. The kernel is object-oriented, and defines a set of classes that represent important entities for a reacting-flow simulation. The kernel is described in more detail elsewhere.

Much of the object-oriented character of the C++ kernel is retained even in languages that are not themselves object-oriented, like Fortran. Fortran 90 actually *does* implement a number of things that allow some degree of object-oriented programming; as a front end for C++ objects it works very well.

From Fortran 90, you can create objects that represent reacting mixtures, or various types of reactors, or ODE integrators, or many other useful things. A great advantage of an object-oriented framework is that objects can be combined together to build more complex dynamic system and process models. For example, it is a simple matter to quickly put together a complex network of stirred reactors, complete with sensors, actuators, and closed-loop controllers, if desired.

1.6 The `cantera` Module

To use Cantera from Fortran 90, put the statement

```
use cantera
```

at the top of any program unit where you need to access a Cantera object, constant, or procedure. Module `cantera` contains the Cantera Fortran 90 interface specification, and handles calling the C++ kernel procedures. It in turn uses othe Cantera modules. You do nor have to include a `use` statement for these others in your program, but the module information files must be available for them.

All module files are located in directory 'Cantera/fortran/modules'. You should either tell your Fortran compiler to look for module files there, or else copy them all to the directory where your application files are.

1.6.1 Declaring an Object

Cantera represents objects in Fortran 90 using Fortran derived types. Derived types are not part of the older Fortran 77, which is one reason why Cantera implements a Fortran 90 interface, but not one for Fortran 77.

A few of the object types Cantera defines are

mixture_t
: Reacting mixtures.

cstr_t
Continuously-stirred tank reactors.

flowdev_t
Flow control devices.

eos_t
Equation of state managers.

kinetics_t
Kinetics managers.

transport_t
Transport managers.

By convention, all type names end in `'_t'`. This serves as a visual reminder that these are object types, and is consistent with the C/C++ naming convention (e.g., `size_t`, `clock_t`).

Objects may be declared using a syntax very similar to that used to declare simple variables:

```
type(mixture_t) mix1, mix2, a, b, c
type(cstr_t) r1, r2, d, e, f
```

The primary difference is that the type name is surrounded by `type(...)`.

1.6.2 Constructing An Object

The objects that are "visible" in your program are really only a Fortran facade covering an object belonging to one of the Cantera C++ classes (which we will call a "kernel object.") The Fortran object is typically very small — in many cases, it holds only a single integer, which is the address of the C++ object, converted to a form representable by a Fortran variable (usually an integer). We call this internally-stored variable the *handle* of the C++ object.

When a Fortran derived-type object is declared, no initialization is done, and there is no underlying C++ object for the handle to point to. Therefore, the first thing you need to do before using an object is call a procedure that constructs an appropriate kernel object, stores its address (converted to an integer) in the Fortran object's handle, and performs any other initialization needed to produce a valid, functioning object.

We refer to such procedures as *constructors*. In Cantera, constructors are Fortran functions that return an object. The syntax to construct an object is shown below.

```
type(mixture_t) gasmix
...
gasmix = GRIMech30()
```

After the object is declared, but before its first use, it is assigned the return value of a constructor function. In this example, constructor `GRIMech30` is called, which returns an object representing a gas mixture that conforms to the widely-used natural gas combustion mechanism GRI-Mech version 3.0 [Smith et al., 1997].

Object construction is always done by assignment, as shown here. The assignment simple copies the handle from the object returned by `GRIMech30()` to object `gasmix`. Once this has been done, the handle inside `gasmix` points to a real C++ object, and it is one that models a gas mixture containing the 53 species and 325 reactions of GRI-Mech 3.0.

We can check this by printing some attributes of the mixture:

```
use cantera
type(mixture_t) mix
character*10 elnames(5), spnames(53)
character*20 rxn
mix = GRIMech30()
write(*,*) nElements(mix), nSpecies(mix), nReactions(mix)
call getElementNames(mix, elnames)
write(*,*) 'elements: '
write(*,*) (elnames(m),m=1,nElements(mix))
call getSpeciesNames(mix, spnames)
write(*,*) 'species: '
write(*,*) (spnames(k),k=1,nSpecies(mix))
write(*,*) 'first 10 reactions:'
do i = 1,10
   call getReactionString(mix, i, rxn)
   write(*,*) rxn
end do
```

The resulting output is

```
           5          53         325
 elements:
 O          H          C          N          AR
 species:
 H2         H          O          O2         OH         H2O        HO2
 H2O2       C          CH         CH2        CH2(S)     CH3        CH4
 CO         CO2        HCO        CH2O       CH2OH      CH3O       CH3OH
 C2H        C2H2       C2H3       C2H4       C2H5       C2H6       HCCO
 CH2CO      HCCOH      N          NH         NH2        NH3        NNH
 NO         NO2        N2O        HNO        CN         HCN        H2CN
 HCNN       HCNO       HOCN       HNCO       NCO        N2         AR
 C3H7       C3H8       CH2CHO     CH3CHO
 first 10 reactions:
 2 O + M <=> O2 + M
 O + H + M <=> OH + M
 O + H2 <=> H + OH
 O + HO2 <=> OH + O2
 O + H2O2 <=> OH + HO
 O + CH <=> H + CO
 O + CH2 <=> H + HCO
 O + CH2(S) <=> H2 +
 O + CH2(S) <=> H + H
 O + CH3 <=> H + CH2O
```

These attributes are in fact correct for GRI-Mech 3.0.

Sometimes the constructor function can fail. For example, constructor `GRIMech30` will fail if you don't have CANTERA_ROOT set properly, since it uses this to find data files it needs to build the mixture.

Cantera implements a `LOGICAL` function `ready` to test whether or not an object is ready to use. Before calling a constructor for some object *objct*, `ready(objct)` returns `.FALSE.`. After successful construction, it returns `.TRUE.`.

Some object constructors also write log files, which may contain warning or error messages. If the construction process fails for no apparent reason, check the log file if one exists.

1.7 Methods

In object-oriented languages, every class of objects has an associated set of *methods* or *member functions* that operate specifically on objects of that class. An object that has a "temperature" attribute might have method `setTemperature(t)` to set the value, and `temperature()` to read it.

The syntax to invoke these methods on an object in C++ or Java would be

```
x.setTemperature(300.0);              // set temp. of x to 300 K
y.setTemperature(x.temperature());    // set y temp. to x temp.
```

Here object `y` might belong to an entirely different class than `x`, and the process of setting its temperature might involve entirely different internal operations. Both classes can define methods with the same name, since it is always clear which one should be called — the one that belongs to the class of the object it is "attached" to.

Fortran is not an object-oriented language, and has only functions and subroutines, not object-specific methods. Therefore, the Cantera Fortran 90 interface uses functions and subroutines to serve the role of methods. Thus, the temperature attribute of an object is retrieved by calling function `temperature(object)`.

> What happens if you try to access a temperature attribute as `object%temperature`? Actually, the compiler will complain that the object has no member `temperature` and will abort. Property values are not stored in the Fortran derived type objects. They are either stored in the kernel object, or computed on-the-fly, depending on which attribute you request.

Mixtures and reactors are two of the Cantera object types that have a temperature attribute. We would like to be able to write something like this:

```
use cantera
type(mixture_t)    mix
type(cstr_t)       reactor
write(*,*) temperature(mix), temperature(reactor)
```

This presents an immediate problem (but one that Fortran 90 fortunately provides a solution for). If we just define function `temperature` in the usual way, we would have to specify the argument type. If the function were then called with an argument of a different type, then (depending on the compiler and compile options) the compile might abort (good), or the compiler might warn you (ok), or it might accept it (bad), resulting in code that runs but gives erroneous results.

To fix this problem, we could define seperate function names for each (`mixture_temperature`, `cstr_temperature`, etc. but this would be awkward at best. (This is precisely what we would have to do in Fortran 77.)

Fortunately, Fortran 90 provides a mechanism so that we can call functions `temperature(object)`, `setTemperature(object,t)` (or any other function) for any type of object. This is done using *generic names*. Cantera really *does* define different function names for each type of object — there is a function `mix_temperature()` that returns the temperature attribute of a `mixture_t` object, an entirely different function `reac_temperature()` that does the same for `cstr_t` objects, and so on. But in module `cantera`, `temperature` is declared to be a generic name that any of these functions can be called by. When a call to `temperature` is made, the compiler looks at the argument list, and then looks to see if any function that takes these argument types has been associated with this generic name. If so, the call to `temperature` is replaced by one to the appropriate function, and if not, a compile error results.

This procedure is analogous to using generic names for intrinsic Fortran function — if you call `sin(x)` with a double precision argument, function `dsin(x)` is actually called; if `x` has type `real(4)`, then function `asin(x)` is called. Fortran 90 simply extends this capability to any function.

1.8 Units

Cantera uses the SI unit system. The SI units for some common quantities are listed in Table 1.8, along with conversion factors to cgs units.

Property	**SI unit**	**cgs unit**	**SI to cgs multiplier**
Temperature	K	K	1
Pressure	Pa	dyne/cm^2	10
Density	kg/m^3	gm/cm^3	0.001
Quantity	kmol	mol	1000
Concentration	kmol/m^3	mol/cm^3	0.001
Viscosity	Pa-s	gm/cm-s	10
Thermal Conductivity	W/m-K	erg/cm-s-K	10^5

1.9 Useful Constants

The `cantera` module defines several useful constants, including the ones listed here.

Constant	**Value**	**Description**
`OneAtm`	1.01325×10^5	Atmospheric pressure in Pascals.
`Avogadro`	$6.022136736 \times 10^{26}$	Number of particles in one kmol.
`GasConstant`	8314.0	The universal ideal gas constant $\hat{R}$.
`StefanBoltz`	5.6705×10^{-8}	The Stephan-Boltzmann constant σ [W/m^2-K^4].
`Boltzmann`	`GasConstant / Avogadro`	Boltzmann's constant k [J/K].

1.10 An Example Program

To see how this all works, let's look at a complete, functioning Cantera Fortran 90 program. This program is a simple chemical equilibrium calculator.

```
      program eq
      use cantera
      implicit double precision (a-h,o-z)
      character*50 xstring
      character*2 propPair
      type(mixture_t) mix

      mix = GRIMech30()
      if (.not.ready(mix)) then
         write(*,*) 'error encountered constructing mixture.'
         stop
      end if

      write(*,*) 'enter T [K] and P [atm]:'
      read(*,*) temp, pres
      pres = OneAtm * pres    ! convert to Pascals

! the initial composition should be entered as a string of name:<moles> pairs
! for example 'CH4:3, O2:2, N2:7.52'. The values will be normalized
! internally
      write(*,*) 'enter initial composition:'
      read(*, 10) xstring
  10  format(a)

! set the initial mixture state
      call setState_TPX(temp, pres, xstring)

! determine what to hold constant
      write(*,*) 'enter one of HP, TP, UV, or SP to specify the
      write(*,*) 'properties to hold constant:'
      read(*,10) propPair

! equilibrate the mixture
      call equilibrate(mix, propPair)

! print a summary of the new state
      call printSummary(mix)

      stop
      end
```

Each of the functions or subroutines used here is described in detail in later chapters.

1.11 Summary

So in summary, the procedure to write a Cantera-based Fortran 90 program is simple:

1. Put the statement `use cantera` at the top of any program unit where you want to use Cantera.
2. *Declare* one or more objects.
3. *Construct* the objects that you declared by assigning them the return value of a *constructor function*.
4. Use the object in your program.

In the following chapters, the objects Cantera implements, their constructors, and their methods will be described in detail. Many examples will also be given. You should try programming a few of these yourself, to verify that you get the same results as shown here. Once you have done that, you're ready to use Cantera for your own applications.

CHAPTER

TWO

Mixtures

Some of the most useful objects Cantera provides are ones representing mixtures. Mixture objects have a modular structure, allowing many different types of mixtures to be simulated by combining different mixture components. You can build a mixture object to suit your needs by selecting an equation of state, a model for homogeneous kinetics (optional), and one for transport properties (also optional). You can then specify the elements that may be present in the mixture, add species composed of these elements, and add reactions among these species. Mixture objects also can be used to model pure substances, by adding only one species to the "mixture."

If you like, you can carry out this process of mixture construction by calling Fortran procedures for each step. But Cantera also provides a set of turn-key constructors that do all of this for you. For example, if you want an object representing pure water (including vapor, liquid, and saturated states), all you need to do is call constructor **`Water()`**. It takes care of installing the right equation of state and installing a single species composed of two atoms of hydrogen and one of oxygen.

Or if you want an object that allows you to do kinetics calculations in in a way that is compatible with the the CHEMKIN™ package, just call constructor **`CKGas`** with your reaction mechanism input file as the argument. **`CKGas`** installs the appropriate kinetics model, equation of state, and pure species property parameterizations, and then adds the elements, species, and reactions specified in your input file. Using `CKGas`, a complete object representing your reaction mechanism, ready for use in kinetics simulations, can be constructed with a single function call.

In this chapter, we will take a first look at objects representing mixtures (and pure substances). We will have much more to say about mixture objects in the next few chapters, where we will cover their thermodynamic properties, kinetic rates, and transport properties.

2.1 Mixture Objects

2.1.1 The Fortran Mixture Object

The basic object type used to represent matter is type `mixture_t`. As this name suggests, Cantera regards all material systems as *mixtures*, although the term is used somewhat loosely, so that this type can represent pure substances too. A pure substance is simply regarded as a mixture that happens to contain only one species.

Objects representing mixtures are implemented in Fortran 90 by the derived type `mixture_t`. Like all Cantera objects, the `mixture_t` object itself is very thin. It contains only 4 integers – the handle that is

used to access the underlying C++ object created by the constructor, the number of elements, the number of species, and the number of reactions. All other information about the mixture is stored in the kernel-level C++ object.

2.1.2 The Kernel Mixture Object

As mentioned above, the kernel mixture object has a modular structure, with interchangeable parts. The basic structure is shown in Figure **??**. A mixture object contains three major components, each of which has specific responsibilities. The *thermodynamic property manager* handles all requests for thermodynamic properties of any type; the *transport property manager* is responsible for everything to do with transport properties, and the *kinetics manager* deals with everything related to homogeneous kinetics. We refer to these three internal objects as the "property managers."

The mixture object *itself* does not compute thermodynamic, transport, or kinetic properties; it simply directs incoming requests for property values to the appropriate manager. It also mediates communication between the property managers — if the transport manager needs a thermodynamic property, the mixture object gets it from the thermodynamic property manager and hands it to the transport manager. The managers have no direct links, making it possible to swap one out for another one of the same type without disrupting the operation of the other property managers.

The mixture object does handle a few things itself. It serves as the interface to the application program, maintains lists of species and element properties, stores and retrieves data specifying the state (internally represented by temperature, density, and mass fractions), and orchestrates property updating when the state has changed.

mixture_t
Object type for mixtures.

nel	integer . Number of elements
nsp	integer . Number of species
nrxn	integer . Number of reactions
hndl	integer . Handle encoding pointer to the C++ object

2.2 Procedures

2.3 Constructors

2.3.1 Reacting Gas Mixtures

To carry out a kinetics simulation with Cantera, an object representing a chemically-reacting mixture is required. While such an object can be created using the basic `Mixture` constructor and explicitly adding all necessary components, it is usually more convenient to call a constructor that does all that for you and returns a finished object, ready to use in simulations.

In this section, constructors that return objects that model reacting gas mixtures are described. These constructors typically parse an input file or files, which specify the elements, species, and reactions to be included, and provide all necessary parameters needed by kinetic, thermodynamic, and transport managers.

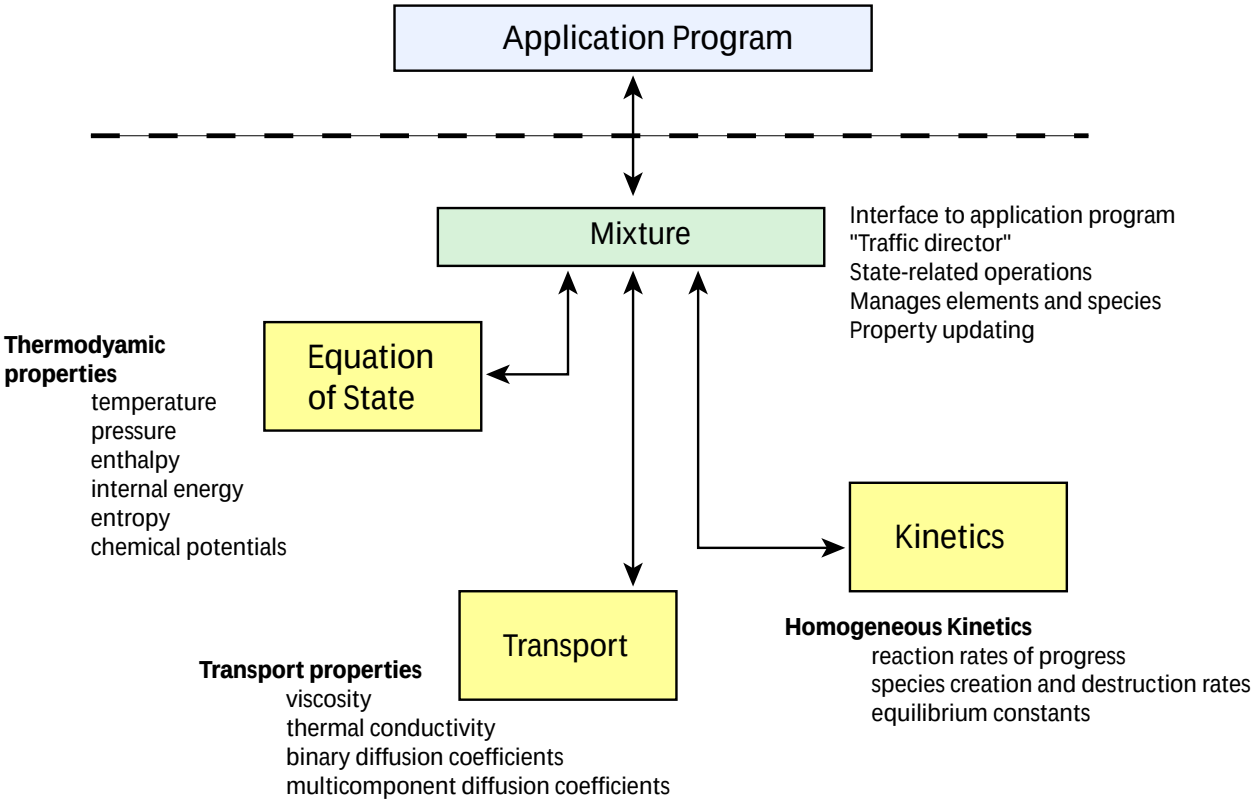

Figure 2.1: Top-level structure of mixture objects. Each

Release Note: In this early release, the only reacting mixture model implemented is one compatible with the model described in Kee et al. [1989]. The capability to define other models is present in the kernel, however.

1) CKGas

Constructor `CKGas` implements a mixture model consistent with that described by Kee et al. [1989]. This model is designed to be used to represent low-density, reacting gas mixtures. The mixture components installed by this constructor are listed below.

Equation of State Manager. The **`IdealGasEOS`** equation of state manager is installed.

Species Thermodynamic Properties. The NASA polynomial parameterization is used to represent the thermodynamic properties of the individual species. See **??**

Kinetics. The **`CK_Kinetics`** kinetics manager is used for homogeneous kinetics. See Chapter **??**

Transport. The **`NullTransport`** transport manager is installed, disabling evaluation of transport properties until another manager is installed by calling **`setTransport`**. Other compatible transport managers include **`MultiTransport`** and **`MixTransport`**.

Homogeneous Kinetics. The *CK_Kinetics* model for homogeneous kinetics is used. See chapter **??** for more information on this kinetics model.

Transport Properties. Transport properties are disabled by default, since many applications do not require them. Any supported transport property model can be used, including *MultiTransport* and *MixTransport*, which are compatible with the multicomponent and mixture-averaged models, respectively, described by Kee et al. [1986], **?**.

Syntax

object = CKGas(*inputFile*, *thermoFile*, *iValidate*)

Arguments

inputFile (Input) Input file. The input file defines the elements, species, and reactions the mixture will contain. It is a text file, and must be in the format[1] specified in Kee et al. [1989]. . This file format is widely used to describe gas-phase reaction mechanisms, and several web sites have reaction mechanisms in CK format available for download. A partial list of such sites is maintained at the Cantera User's Group site http://groups.yahoo.com/group/cantera. If omitted, an empty mixture object is returned.

Cantera actually defines several extensions to the format of Kee et al. [1989]. See Appendix **??** for a complete description of CK format with Cantera extensions. [character*(*)].

thermoFile (Input) CK-format species property database. Species property data may be specified either in the `THERMO` block of the input file, or in *thermoFile*. If species data is present in both files for some species, the input file takes precedence. Note that while *thermoFile* may be any CK-format file, only its `THERMO` section will be read. If *thermoFile* is an empty string, then all species data must be contained in the input file. [character*(*)].

[1] We refer to this format as "CK format"

iValidate (Input) This flag determines the degree to which the reaction mechanism is validated as the file is parsed. If omitted or zero, only basic validation that can be performed quickly is done. If non-zero, extensive and possibly slow validation of the mechanism will be performed. [integer].

Result

Returns an object of type **mixture_t**.

Example

```
      type(mixture_t) mix1, mix2, mix3
!  all species data in 'mymech.inp'
      mix1 = CKGas('mymech.inp')
!  look for species data in 'thrm.dat' if not in 'mech2.inp'
      mix2 = CKGas('mech2.inp', 'thrm.dat')
!  turn on extra mechanism validation
      mix3 = CKGas('mech3.inp', '', 1)
```

See Also

addDirectory, GRIMech30

2) GRIMech30

This constructor builds a mixture object corresponding to the widely-used natural gas combustion mechanism GRI-Mech version 3.0 [Smith et al., 1997]. The mixture object created contains 5 elements, 53 species, and 325 reactions.

Syntax

object = GRIMech30()

Result

Returns an object of type **mixture_t**.

Description

The natural gas combustion mechanism GRI-Mech version 3.0 Smith et al. [1997] is widely-used for general combustion calculations. [more...]

At present, this constructor simply calls the `CKGas` constructor with input file 'grimech30.inp'. This file is located in directory 'data/CK/mechanisms' within the 'cantera-1.2' tree. For this to work correctly, environment variable `CANTERA_ROOT` must be set as described in Chapter 1.

In a future relese, calling this constructor will generate a hard-coded version of GRI-Mech 3.0, that may be faster than the one generated by calling **CKGas**.

For more information on GRI-Mech, see http://www.me.berkeley.edu/gri_mech

Example

```
type(mixture_t) mix
mix1 = GRIMech30()
```

2.3.2 Pure Substance Constructors

In addition to the ideal gas mixtures discussed above, Cantera implements several non-ideal pure fluids. These use accurate equations of state, and are valid in the vapor, liquid, and mixed liquid/vapor regions of the phase diagram. In most cases, the equation of state parameterizations are taken from the compilation by Reynolds [Reynolds, 1979].

The standard-state enthalpy of formation and absolute entropy have been adjusted to match the values in the JANAF tables. These objects should therefore be thermodymamically consistent with most gas mixture objects that use thermodynamic data obtained from fits to JANAF data.

3) Water

Constructs an object representing pure water, H_2O.

Syntax

object = Water()

Result

Returns an object of type **mixture_t**.

Description

The equation of state parameterization is taken from the compilation by Reynolds [1979]. The original source is **?**. The standard-state enthalpy of formation and the absolute entropy have been adjusted to match the values in the JANAF tables.

Example

```
type(substance_t) h2o
h2o = Water()
```

4) Nitrogen

Construct an object representing nitrogen, N_2.

Syntax

object = Nitrogen()

Result

Returns an object of type **mixture_t**.

Description

The equation of state parameterization is taken from the compilation by Reynolds [1979]. The original source is Jacobsen et al. [1972]. The standard-state enthalpy of formation and the absolute entropy have been adjusted to match the values in the JANAF tables.

Example

```
type(substance_t) subst
subst = Nitrogen()
```

5) Methane

Construct a pure substance object representing methane, CH_4.

Syntax

object = Methane()

Result

Returns an object of type **mixture_t**.

Description

The equation of state parameterization is taken from the compilation by Reynolds [1979]. The original source is **?**. The standard-state enthalpy of formation and the absolute entropy have been adjusted to match the values in the JANAF tables.

Example

```
type(substance_t) subst
subst = Methane()
```

6) Hydrogen

Construct a pure substance object representing hydrogen, H_2.

Syntax

object = Hydrogen()

Result

Returns an object of type **mixture_t**.

Description

The equation of state parameterization is taken from the compilation by Reynolds [1979]. The original source is **?**. The standard-state enthalpy of formation and the absolute entropy have been adjusted to match the values in the JANAF tables.

Example

```
type(substance_t) subst
subst = Hydrogen()
```

7) Oxygen

Construct a pure substance object representing oxygen, O_2.

Syntax

object = Oxygen()

Result

Returns an object of type **mixture_t**.

Description

The equation of state parameterization is taken from the compilation by Reynolds [1979]. The original source is **?**. The standard-state enthalpy of formation and the absolute entropy have been adjusted to match the values in the JANAF tables.

Example

```
type(substance_t) subst
subst = Oxygen()
```

Example 2.1

Constant-pressure lines for nitrogen are shown on a volume-temperature plot in Fig. 2.2. The data for this plot were generated by the program below. Note that the state of the nitrogen object may correspond to liquid, vapor, or saturated liquid/vapor.

```
program plotn2
use cantera
implicit double precision (a-h,o-z)
double precision pres(5),v(5)
data pres/1.d0, 3.d0, 10.d0, 30.d0, 100.d0/
type(substance_t) s
s = Nitrogen()
open(10,file='plot.csv',form='formatted')
t = minTemp(s)
do n = 1,100
   t = t + 1.0
   do j = 1,5
      call setState_TP(s, t, OneAtm*pres(j))
      v(j) = 1.0/density(s)
   end do
```

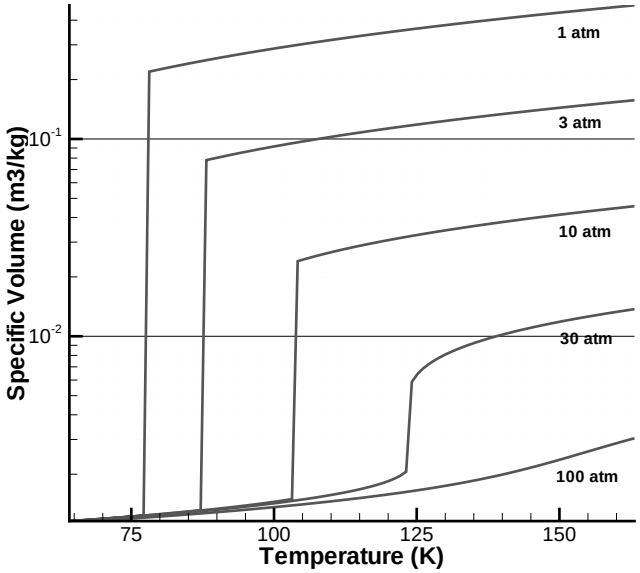

Figure 2.2: Isobars at 1, 3, 10, 30, and 100 atm for nitrogen.

```
          write(10,20) t,(v(j), j = 1,5)
20        format(6(e14.5,', '))
      end do
      close(10)
      stop
      end
```

2.4 Utilities

The procedures in this section provide various utility functions.

8) addDirectory

Add a directory to the path to be searched for input files.

Syntax

call addDirectory(*dirname*)

Arguments

dirname (Input) Directory name [character*(*)].

Description

Constructor functions may require data in external files. Cantera searches for these files first in the local directory. If not found there and environment variable $CANTERA_DATA_DIR is set to a directory name, then this directory will be searched. Additional directories may be added to the search path by calling this subroutine.

Example

```
call addDirectory('/usr/local/my_filea')
```

9) delete

Deletes the kernel mixture object.

Syntax

call delete(*mix*)

Arguments

mix (Input) Mixture object [type(mixture_t)].

Description

When a mixture constructor is called, a kernel mixture object is dynamically allocated. This object stores all data required to compute the mixture thermodynamic, kinetic, and transport properties. For mixtures of many species, or ones with a large reaction mechanism, the size of the object may be large. Calling this subroutine frees this memory. After deleting a mixture, the object must be constructed again before using it.

10) ready

Returns true if the mixture has been successfully constructed and is ready for use.

Syntax

result = ready(*mix*)

Arguments

mix (Input) Mixture object [type(mixture_t)].

Result

Returns a `logical` value.

11) copy

Copies one mixture object to another.

Syntax

call copy(*dest*, *src*)

Arguments

dest	(Output) Mixture object [type(mixture_t)].
src	(Input) Mixture object [type(mixture_t)].

Description

Performs a "shallow" copy of one mixture object to another. Note that the kernel object itself is *not* copied, only the handle. Therefore, after calling `copy`, both objects point to the same kernel object. The contents of *dest* are overwritten. Assigning one mixture to another invokes this procedure.

Example

```
use cantera
type(mixture_t) fluid1, fluid2, fluid3
fluid1 = Water()
fluid2 = Methane()
write(*,*) 'fluid1 ready: ',ready(fluid1)
call delete(fluid1)       ! fluid1 is now empty
write(*,*) 'fluid1 ready: ',ready(fluid1)
call copy(fluid1, fluid2) ! now fluid1 is Methane too
write(*,*) 'fluid1 ready: ',ready(fluid1)
fluid3 = fluid2           ! assignment calls 'copy'
write(*,*) critTemperature(fluid1)
write(*,*) critTemperature(fluid2)
write(*,*) critTemperature(fluid3)
```

Output:

```
 fluid1 ready:  T
 fluid1 ready:  F
 fluid1 ready:  T
   190.555000000000
   190.555000000000
   190.555000000000
```

12) saveState

Write the interal thermodynamic state information to an array.

Syntax

call saveState(*mix*, *state*)

Arguments

mix (Input) Mixture object [type(mixture_t)].

state (Output) Array where state information will be stored [double precision array]. Must be dimensioned at least as large as the number of species in *mix* + *2*.

Description

calling **saveState** writes the internal data defining the thermodynamic state of a mixture object to an array. It can later be restored by calling **restoreState**.

Example

```
use cantera
real(8) state(60) ! dimension at least (# of species) + 2
type(mixture_t) mix
mix = GRIMech30()
call setState_TPX(mix, 400.d0, OneAtm, 'N2:0.2, H2:0.8')
write(*,*) entropy_mole(mix), moleFraction(mix,'N2')
call saveState(mix, state)
call setState_TPX(mix, 1000.d0, 1.d3, 'N2:0.8, CH4:0.2')
write(*,*) entropy_mole(mix), moleFraction(mix,'N2')
call restoreState(mix, state)
write(*,*) entropy_mole(mix), moleFraction(mix,'N2')
```

Output:

```
 155555.372874368       0.200000000000000
 236274.031245184       0.800000000000000
 155555.372874368       0.200000000000000
```

13) restoreState

Restore the mixture state from the data in array *state*. written by a prior call to *saveState*.

Syntax

call restoreState(*mix*, *state*)

Arguments

mix (Input) Mixture object [type(mixture_t)].

state (Output) Array where state information will be stored [double precision array]. Must be dimensioned at least as large as the number of species in *mix* + *2*.

Example see the example for procedure 12.

2.5 Number of Elements, Species, and Reactions

14) nElements

Number of elements.

Syntax

result = nElements(*mix*)

Arguments

mix (Input) Mixture object [type(mixture_t)].

Result

Returns a `integer` value.

Example

```
nel = nElements(mix)
```

15) nSpecies

Number of species.

Syntax

result = nSpecies(*mix*)

Arguments

mix (Input) Mixture object [type(mixture_t)].

Result

Returns a `integer` value.

Example

```
use cantera
type(mixture_t) mix
mix = GRIMech30()
write(*,*) 'number of species = ',nSpecies(mix)
```

Output:

```
number of species = 53
```

16) nReactions

Number of reactions.

Syntax

result = nReactions(*mix*)

Arguments

mix (Input) Mixture object [type(mixture_t)].

Result

Returns a `integer` value.

Example

```
use cantera
type(mixture_t) mix
mix = GRIMech30()
write(*,*) 'number of reactions = ',nReactions(mix)
```

Output:

```
number of reactions = 325
```

2.6 Mixture Creation

The procedures in this section are designed to allow creation of arbitrary custom mixtures from Fortran. To use these procedures, first construct a skeleton mixture object using constructor **`Mixture`**. This returns an "empty" mixture — one that has no constituents, and with do-nothing "null" property managers installed. By adding elements and species, and installing managers for thermodynamic properties, kinetics, and transport properties, arbitrary mixtures can be constructed.

Release Note: These capabilities are not yet fully functional in the Fortran yet. More procedures will be added soon.

17) Mixture

Construct an empty mixture object with no constituents or property managers.

Syntax

object = Mixture()

Result

Returns an object of type **`mixture_t`**.

Description

This constructor creates a skeleton object, to which individual components may be added by calling the procedures below.

Example

```
type(mixture_t) mix
mix = Mixture()
```

18) addElement

Add an element to the mixture.

Syntax

call addElement(*mix*, *name*, *atomicWt*)

Arguments

mix	(Input) Mixture object [type(mixture_t)].
name	(Input) Element name [character*(*)].
atomicWt	(Input) Atomic weight in amu [double precision].

Example

```
type(mixture_t) mix
mix = Mixture()
call addElement('Si', 28.0855)
call addElement('H', 1.00797)
...
!  add species, reactions, etc. (not yet implemented)
...
```

2.7 Properties of the Elements

The *elements* in a mixture are the entities that species are composed from, and must be conserved by reactions. Usually these correspond to elements of the periodic table, or one of their isotopes. However, for reactions involving charged species, it is useful to define some charged entity (usually an electron) as an element, to insure that all reactions conserve charge.

19) getElementNames

Get the names of the elements.

Syntax

call getElementNames(*mix, enames*)

Arguments

mix (Input) Mixture object [type(mixture_t)].

enames (Output) Element names [character*(*) array]. Must be dimensioned at least as large as the number of elements in *mix*.

Description

The element names may be strings of any length. If they names are longer than the number of characters in each entry of the *enames* array, the returned names will be truncated.

20) elementIndex

Index number of an element.

Syntax

result = elementIndex(*mix, name*)

Arguments

mix (Input) Mixture object [type(mixture_t)].

name (Input) Element name [character*(*)].

Result

Returns a `integer` value.

If the element is not present in the mixture, the value -1 is returned.

Description

The index number corresponds to the order in which the element was added to the mixture. The first element added has index number 1, the second has index number 2, and so on. All arrays of element properties (e.g., names, atomic weights) are ordered by index number.

Example

```
! v1 contains element properties ordered as listed in mech1.inp.
! copy these values to the the positions in array v2 which is
! ordered as in 'mech2.inp'
      type(mixture_t) mix, mix2
      mix1 = CKGas('mech1.inp')
      mix2 = CKGas('mech2.inp')
      do i = 1,nElements(mix1)
         ename = elementName(i)
         loc2 = elementIndex(mix2, ename)
         v2[loc2] = v1[i]
      end do
```

See Also
speciesIndex

21) atomicWeight

The atomic weight of the m^{th} element [kg/kmol].

Syntax
result = atomicWeight(*mix*, *m*)

Arguments

mix	(Input) Mixture object [type(mixture_t)].
m	(Input) Element index [integer].

Result
Returns a `double precision` value.

Example

```
character(20) names(10)
call getElementNames(mix, names)
do m = 1, nElements(mix)
  write(*,*) names(m), atomicWeight(mix, m)
end do
```

2.8 Properties of the Species

The *species* are defined so that any achievable chemical composition for the mixture can be represented by specifying the mole fraction of each species. In gases, the species generally correspond to distinct molecules, atoms, or ions, and each is composed of integral numbers of atoms of the elements.

However, if the gas mixture contains molecules with populated metastable states, then to fully represent any possible mixture composition, these states (or groups of them) must also be defined as species. Common examples of such long-lived metastable states include singlet Δ oxygen and singlet methylene.

In solid or liquid solutions, it may be necessary to define species composed of non-integral numbers of elements. This is acceptable, as long as all realizable states of the mixture correspond to some set of mole fractions for the species so defined.

22) nAtoms

Number of atoms of element *m* in species *k*

Syntax

result = nAtoms(*mix*, *k*, *m*)

Arguments

mix	(Input) Mixture object [type(mixture_t)].
k	(Input) Species index [integer].
m	(Input) Element index [integer].

Result

Returns a `double precision` value.

Non-integral and/or negative values are allowed. For example, if an electron is defined to be an element, then a positive ion has a negative number of "atoms" of this element.

23) charge

Electrical charge of the k^{th} species, in multiples of e, the magnitude of the electron charge.

Syntax

result = charge(*mix*, *k*)

Arguments

mix	(Input / Output) Mixture object [type(mixture_t)].
k	(Input) Species index [integer].

Result

Returns a `double precision` value.

In most cases, the charge is integral, but non-integral values are allowed.

24) speciesIndex

Index number of a species.

Syntax

result = speciesIndex(*mix*, *name*)

Arguments

mix	(Input) Mixture object [type(mixture_t)].
name	(Input) Species name [character*(*)].

Result

Returns a `integer` value.

Description

The index number corresponds to the order in which the species was added to the mixture. All species property arrays are ordered by index number.

25) getSpeciesNames

Get the array of species names.

Syntax

call getSpeciesNames(*mix*, *snames*)

Arguments

mix (Input) Mixture object [type(mixture_t)].

snames (Output) Array of species names [character*(*) array]. Must be dimensioned at least as large as the number of species in *mix*.

Description

The species names may be strings of any length. If the species names are longer than the number of characters in each element of the *snames* array, the returned names will be truncated.

26) molecularWeight

The molecular weight of species *k* [amu].

Syntax

result = molecularWeight(*mix*, *k*)

Arguments

mix (Input) Mixture object [type(mixture_t)].

k (Input) Species index [integer].

Result

Returns a `double precision` value.

27) getMolecularWeights

Get the array of species molecular weights.

Syntax

call getMolecularWeights(*mix*, *molWts*)

Arguments

mix (Input / Output) Mixture object [type(mixture_t)].

molWts (Output) Species molecular weights [double precision array]. Must be dimensioned at least as large as the number of species in *mix*.

2.9 Mixture Composition

The mixture composition may be specified by mole fractions, mass fractions, or concentrations. The procedures described here set or get the mixture composition. Additional procedures that set the composition are described in the next chapter.

28) setMoleFractions

Set the species mole fractions, holding the temperature and mass density fixed.

Syntax

call setMoleFractions(*mix*, *moleFracs*)

Arguments

mix (Input / Output) Mixture object [type(mixture_t)].

moleFracs (Input) Species mole fractions [double precision array]. Must be dimensioned at least as large as the number of species in *mix*.

29) setMoleFractions_NoNorm

Set the species mole fractions *without* normalizing them to sum to 1.

Syntax

call setMoleFractions_NoNorm(*mix*, *moleFracs*)

Arguments

mix (Input / Output) Mixture object [type(mixture_t)].

moleFracs (Input) Species mole fractions [double precision array]. Must be dimensioned at least as large as the number of species in *mix*.

Description

This procedure is designed for use when perturbing the composition, for example when computing Jacobians.

30) setMassFractions

Set the species mass fractions, holding the temperature and mass density fixed.

Syntax

call setMassFractions(*mix*, *massFracs*)

Arguments

mix (Input / Output) Mixture object [type(mixture_t)].

massFracs (Input) Species mass fractions [double precision array]. Must be dimensioned at least as large as the number of species in *mix*.

31) setMassFractions_NoNorm

Set the species mass fractions *without* normalizing them to sum to 1.

Syntax

call setMassFractions_NoNorm(*mix*, *y*)

Arguments

mix (Input / Output) Mixture object [type(mixture_t)].

y (Input) Species mass fractions [double precision array]. Must be dimensioned at least as large as the number of species in *mix*.

Description

This procedure is designed for use when perturbing the composition, for example when computing Jacobians.

32) setConcentrations

Set the species concentrations, holding the temperature fixed.

Syntax

call setConcentrations(*mix*, *conc*)

Arguments

mix (Input / Output) Mixture object [type(mixture_t)].

conc (Input) Species concentrations [kmol/m^3] [double precision array]. Must be dimensioned at least as large as the number of species in *mix*.

33) moleFraction

Mole fraction of one species.

Syntax

result = moleFraction(*mix*, *name*)

Arguments

mix (Input) Mixture object [type(mixture_t)].

name (Input) Species name [character*(*)].

Result

Returns a `double precision` value.

Description

The mole fraction of the species with name *name* is returned. If no species in the mixture has this name, the value zero is returned.

34) massFraction

Mass fraction of one species.

Syntax

result = massFraction(*mix*, *name*)

Arguments

mix (Input) Mixture object [type(mixture_t)].

name (Input) Species name [character*(*)].

Result

Returns a `double precision` value.

Description

The mass fraction of the species with name *name* is returned. If no species in the mixture has this name, the value zero is returned.

35) getMoleFractions

Get the array of species mole fractions.

Syntax

call getMoleFractions(*mix*, *moleFracs*)

Arguments

mix (Input) Mixture object [type(mixture_t)].

moleFracs (Input / Output) Species mole fractions [double precision array]. Must be dimensioned at least as large as the number of species in *mix*.

36) getMassFractions

Get the array of species mass fractions.

Syntax

call getMassFractions(*mix*, *massFracs*)

Arguments

mix (Input) Mixture object [type(mixture_t)].

massFracs (Input / Output) Species mass fractions [double precision array]. Must be dimensioned at least as large as the number of species in *mix*.

37) getConcentrations

Get the array of species concentrations.

Syntax

call getConcentrations(*mix, concentrations*)

Arguments

mix (Input) Mixture object [type(mixture_t)].

concentrations (Input / Output) Species concentrations. [kmol/m^3] [double precision array]. Must be dimensioned at least as large as the number of species in *mix*.

2.10 Mean Properties

These procedures return weighted averages of pure-species properties.

38) mean_X

Return $\sum_k Q_k X_k$, the mole-fraction-weighted mean value of pure species molar property Q.

Syntax

result = mean_X(*mix, Q*)

Arguments

mix (Input) Mixture object [type(mixture_t)].

Q (Input) Array of species molar property values [double precision array]. Must be dimensioned at least as large as the number of species in *mix*.

Result

Returns a `double precision` value.

39) mean_Y

Return $\sum_k Q_k Y_k$, the mass-fraction-weighted mean value of pure species specific property Q.

Syntax

result = mean_Y(*mix, Q*)

Arguments

mix (Input) Mixture object [type(mixture_t)].

Q (Input) Array of species specific property values [double precision array]. Must be dimensioned at least as large as the number of species in *mix*.

Result

Returns a `double precision` value.

40) meanMolecularWeight

The mean molecular weight [amu].

Syntax

result = meanMolecularWeight(*mix*)

Arguments

mix (Input) Mixture object [type(mixture_t)].

Result

Returns a `double precision` value.

Description

The mean molecular weight is defined by

$$\overline{M} = \sum_k M_k X_k. \tag{2.1}$$

An equivalent alternative expression is

$$\frac{1}{\overline{M}} = \sum_k \frac{Y_k}{M_k} \tag{2.2}$$

See Also

molecularWeight

41) sum_xlogQ

Given a vector Q, returns $\sum_k X_k \log Q_k$.

Syntax

result = sum_xlogQ(*mix, Q*)

Arguments

mix (Input) Mixture object [type(mixture_t)].

Q (Output) Array of species property values [double precision array]. Must be dimensioned at least as large as the number of species in *mix*.

Result

Returns a `double precision` value.

This box will be replaced by an introduction discussing equations of state, models implemented in Cantera, etc...

In the last chapter, we saw how to create a basic mixture object. In this chapter, we'll look at how to set the thermodynamic state in various ways, and compute various thermodynamic properties.

eos_t

hndl integer . Handle encoding pointer to the C++ object

2.11 Procedures

2.12 The Thermodynamic State

Although a long list of properties of a mixture can be written down, most of them are not independent — once a few values have been specified, all the rest are fixed.

The number of independent properties is equal to the number of different ways any one property can be altered. If we consider the Gibbs equation written for dU

$$dU = TdS - PdV + \sum_k \mu_k N_k, \tag{2.3}$$

we see that U can be changed by heat addition (TdS) or by compression ($-PdV$), or by changing any one of K mole numbers. Therefore, the number of degrees of freedom is $K + 2$.

However, if we are interested only in the intensive state, there are only $K + 1$ independently variable parameters, since one degree of freedom simply sets the total mass or number of moles.

The *thermodynamic state* is determined by specifying any $K + 1$ *independent* property values. Of these, $K - 1$ specify the composition, and are usually taken to be either mole fractions or mass fractions, although the chemical potentials can be used too. The two remaining can be any two independent mixture properties.

These procedures described here set the thermodynamic state of a mixture, or modify its current state.

Procedures that set the mole fractions or mass fractions take an array of K values, even though only $K - 1$ are independent. Unless otherwise noted, these input values will be internally normalized to satisfy

$$\sum_k X_k = 1 \tag{2.4}$$

or

$$\sum_k Y_k = 1. \tag{2.5}$$

For those procedures that take as an argument the internal energy, enthalpy, or entropy, the value per unit mass must be used.

2.12.1 Setting the State

The procedures described below set the full thermodynamic state of a mixture. They take as arguments two thermodynamic properties and an array or string that specifies the composition. The state of the mixture after calling any of these is independent of its state prior to the call.

42) setState_TPX

Set the temperature, pressure, and mole fractions.

Syntax

call setState_TPX(*mix*, *t*, *p*, *moleFracs*)

Arguments

mix	(Input / Output) Mixture object [type(mixture_t)].
t	(Input) Temperature [K] [double precision].
p	(Input) Pressure [Pa] [double precision].
moleFracs	(Input) Species mole fractions [double precision array]. Must be dimensioned at least as large as the number of species in *mix*.

Description

There are two versions of **`setState_TPX`**. This one uses an array to specify the mole fractions, and the other (below) uses a string.

Example

```
double precision x(100)
...
t = 1700.d0
p = 0.1 * OneAtm
do k=1,nSpecies(mix)
    x(k) = 0.d0
end do
i_nh3 = speciesIndex(mix, 'NH3')
i_h2  = speciesIndex(mix, 'H2')
x(i_nh3) = 2.d0
x(i_h2)  = 1.d0
call setState_TPX(mix, t, p, x)
```

43) setState_TPX

Set the temperature, pressure, and mole fractions.

Syntax

call setState_TPX(*mix*, *t*, *p*, *moleFracs*)

Arguments

mix	(Input / Output) Mixture object [type(mixture_t)].
t	(Input) Temperature [K] [double precision].
p	(Input) Pressure [Pa] [double precision].
moleFracs	(Input) Species mole fraction string [character*(*)].

Description

There are two versions of **`setState_TPX`**. This one uses a string to specify the mole fractions, and the other (above) uses an array. The string should contain comma-separated name/value pairs, each of which is separated by a colon, such as 'CH4:1, O2:2, N2:7.52'. The values do not need to sum to 1.0 - they will be internally normalized. The mole fractions of all other species in the mixture are set to zero.

Example

```
real(8) t, p
character(50) x
t = 300.d0
p = OneAtm
x = 'H2:3, SIH4:0.1, AR:60'
call setState_TPX(t, p, x)
```

44) setState_TPY

Set the temperature, pressure, and mass fractions.

Syntax

call setState_TPY(*mix*, *t*, *p*, *massFracs*)

Arguments

mix	(Input / Output) Mixture object [type(mixture_t)].
t	(Input) Temperature [K] [double precision].
p	(Input) Pressure [Pa] [double precision].
massFracs	(Input) Species mass fractions [double precision array]. Must be dimensioned at least as large as the number of species in *mix*.

Description

There are two versions of **`setState_TPY`**. This one uses an array to specify the mass fractions, and the other (below) uses a string.

45) setState_TPY

Set the temperature, pressure, and mass fractions.

Syntax

call setState_TPY(*mix*, *t*, *p*, *massFracs*)

Arguments

mix	(Input / Output) Mixture object [type(mixture_t)].
t	(Input) Temperature [K] [double precision].
p	(Input) Pressure [Pa] [double precision].
massFracs	(Input) Species mass fraction string [character*(*)].

Description

There are two versions of **`setState_TPY`**. This one uses a string to specify the mole fractions, and the other (above) uses an array. The string should contain comma-separated name/value pairs, each of which is separated by a colon, such as 'CH4:2, O2:1, N2:10' (the values do not need to sum to 1.0 - they will be internally normalized). The mass fractions of all other species in the mixture will be set to zero.

Example

```
real(8) t, p
character(50) x
t = 300.d0
p = OneAtm
x = 'H2O:5, NH3:0.1, N2:20'
call setState_TPY(t, p, x)
```

46) setState_TRX

Set the temperature, density, and mole fractions.

Syntax

call setState_TRX(*mix*, *t*, *density*, *moleFracs*)

Arguments

mix	(Input / Output) Mixture object [type(mixture_t)].
t	(Input) Temperature [K] [double precision].
density	(Input) Density [kg/m^3] [double precision].
moleFracs	(Input) Species mole fractions [double precision array]. Must be dimensioned at least as large as the number of species in *mix*.

47) setState_TRY

Set the temperature, density, and mass fractions.

Syntax

call setState_TRY(*mix*, *t*, *density*, *massFracs*)

Arguments

mix	(Input / Output) Mixture object [type(mixture_t)].
t	(Input) Temperature [K] [double precision].
density	(Input) Density [kg/m^3] [double precision].
massFracs	(Input) Species mass fractions [double precision array]. Must be dimensioned at least as large as the number of species in *mix*.

2.12.2 Modifying the Current State

The procedures in this section set only one or two properties, leaving the temperature, density, and/or composition unchanged.

48) setTemperature

Set the temperature, holding the mass density and composition fixed.

Syntax

call setTemperature(*mix*, *temp*)

Arguments

mix	(Input / Output) Mixture object [type(mixture_t)].
temp	(Input) Temperature [K]. [double precision].

49) setDensity

Set the density, holding the temperature and composition fixed.

Syntax

call setDensity(*mix*, *density*)

Arguments

mix	(Input / Output) Mixture object [type(mixture_t)].
density	(Input) Density [kg/m^3] [double precision].

50) setState_TP

Set the temperature and pressure, holding the composition fixed.

Syntax

call setState_TP(*mix, t, p*)

Arguments

mix	(Input / Output) Mixture object [type(mixture_t)].
t	(Input) Temperature [K] [double precision].
p	(Input) Pressure [Pa] [double precision].

51) setState_PX

Set the pressure and mole fractions, holding the temperature fixed.

Syntax

call setState_PX(*mix, p, x*)

Arguments

mix	(Input / Output) Mixture object [type(mixture_t)].
p	(Input) Pressure [Pa] [double precision].
x	(Input) Species mole fractions [double precision array]. Must be dimensioned at least as large as the number of species in *mix*.

52) setState_PY

Set the pressure and mass fractions, holding the temperature fixed.

Syntax

call setState_PY(*mix, p, massFracs*)

Arguments

mix	(Input / Output) Mixture object [type(mixture_t)].
p	(Input) Pressure [Pa] [double precision].
massFracs	(Input) Species mass fractions [double precision array]. Must be dimensioned at least as large as the number of species in *mix*.

53) setPressure

Set the pressure, holding the temperature and composition fixed.

Syntax

call setPressure(*mix*, *p*)

Arguments

mix	(Input / Output) Mixture object [type(mixture_t)].
p	(Input) Pressure [Pa] [double precision].

54) setState_TR

Set the temperature and density, holding the composition fixed.

Syntax

call setState_TR(*mix*, *t*, *rho*)

Arguments

mix	(Input / Output) Mixture object [type(mixture_t)].
t	(Input) Temperature [K] [double precision].
rho	(Input) Density [kg/m^3] [double precision].

55) setState_TX

Set the temperature and mole fractions, holding the density fixed.

Syntax

call setState_TX(*mix*, *t*, *moleFracs*)

Arguments

mix	(Input / Output) Mixture object [type(mixture_t)].
t	(Input) Temperature [K] [double precision].
moleFracs	(Input / Output) Species mole fractions [double precision array]. Must be dimensioned at least as large as the number of species in *mix*.

56) setState_TY

Set the temperature and mass fractions, holding the density fixed.

Syntax

call setState_TY(*mix*, *t*, *massFracs*)

Arguments

mix (Input / Output) Mixture object [type(mixture_t)].

t (Input) Temperature [K] [double precision].

massFracs (Input) Species mass fractions [double precision array]. Must be dimensioned at least as large as the number of species in *mix*.

57) setState_RX

Set the density and mole fractions, holding the temperature fixed.

Syntax

call setState_RX(*mix*, *density*, *moleFracs*)

Arguments

mix (Input / Output) Mixture object [type(mixture_t)].

density (Input) Density [kg/m^3] [double precision].

moleFracs (Input) Species mole fractions [double precision array]. Must be dimensioned at least as large as the number of species in *mix*.

58) setState_RY

Set the density and mass fractions, holding the temperature fixed.

Syntax

call setState_RY(*mix*, *density*, *y*)

Arguments

mix (Input / Output) Mixture object [type(mixture_t)].

density (Input) Density [kg/m^3] [double precision].

y (Input) Species mass fractions [double precision array]. Must be dimensioned at least as large as the number of species in *mix*.

59) setState_HP

Set the specific enthalpy and pressure, holding the composition fixed.

Syntax

call setState_HP(*mix*, *t*, *p*)

Arguments

mix	(Input / Output) Mixture object [type(mixture_t)].
t	(Input) Specific enthaply [J/kg] [double precision].
p	(Input) Pressure [Pa] [double precision].

Description

This procedure uses a Newton method to find the temperature at which the specific enthalpy has the desired value. The iteration proceeds until the temperature changes by less than 0.001 K. At this point, the state is set to the new temperature value, and the procedure returns. The iterations are done at constant pressure and composition.

This algorithm always takes at least one Newton step. Therefore, small perturbations to the input enthalpy should produce a first-order change in temperature, even if it is well below the temperature error tolerance, allowing its use in routines that calculate derivatives or Jacobians. To test this, in the example below, the specific heat is calculated numerically using this procedure, and compared to the value obtained directly from its parameterization (here a polynomial). (Note that this is not the most direct way to calculate c_p numerically – normally, T would be perturbed, not h.)

Example

```
use cantera
real(8) h0, t1, t2, state(100)
type(mixture_t) mix
mix = GRIMech30()

call setState_TPX(mix,2000.d0,OneAtm,'CH4:1,O2:2')
call saveState(mix, state)
h0 = enthalpy_mass(mix)
call setState_HP(mix, h0, OneAtm)
t1 = temperature(mix)
call setState_HP(mix, h0 + h0*1.d-8, OneAtm)
t2 = temperature(mix)
call restoreState(mix, state)
write(*,*) (h0*1.d-8)/(t2 - t1), cp_mass(mix)
return
```

Output:

```
2199.12043414729        2199.12045953925
```

60) setState_UV

Set the specific internal energy and specific volume, holding the composition constant.

Syntax

call setState_UV(*mix*, *t*, *p*)

Arguments

mix	(Input / Output) Mixture object [type(mixture_t)].
t	(Input) Specific internal energy [J/kg] [double precision].
p	(Input) Specific volume [m^3/kg] [double precision].

Description

See the discussion of procedure **`setState_HP`**, procedure number 59.

61) setState_SP

Set the specific entropy and pressure, holding the composition constant.

Syntax

call setState_SP(*mix*, *t*, *p*)

Arguments

mix	(Input / Output) Mixture object [type(mixture_t)].
t	(Input) Specific entropy [J/kg/K] [double precision].
p	(Input) Pressure [Pa] [double precision].

Description

See the discussion of procedure **`setState_HP`**, procedure number 59.

62) setState_SV

Set the specific entropy and specific volume, holding the composition constant.

Syntax

call setState_SV(*mix*, *t*, *p*)

Arguments

mix	(Input / Output) Mixture object [type(mixture_t)].

t (Input) Specific entropy [J/kg/K] [double precision].

p (Input) Specific volume [m^3/kg] [double precision].

Description

See the discussion of procedure **`setState_HP`**, procedure number 59.

2.13 Species Ideal Gas Properties

The procedures in this section return non-dimensional ideal gas thermodynamic properties of the pure species at the standard-state pressure P_0 and the mixture temperature T.As discussed in Section **??**, the properties of any non-ideal mixture can be determined given the mechanical equation of state $P(T, \rho)$ and the pure-species properties evaluated for the low-density, ideal-gas limit. The procedures below return arrays of pure species ideal gas properties. For those that depend on $\ln \rho$, the values are scaled to a density $\rho = P_0/RT$, where P_0 is the standard-state pressure.

63) getEnthalpy_RT

Get the array of pure-species non-dimensional enthalpies

$$\hat{h}_k(T, P_0)/\hat{R}T, \quad k = 1 \ldots K. \tag{2.6}$$

Syntax

call getEnthalpy_RT(*mix*, *h_RT*)

Arguments

mix (Input) Mixture object [type(mixture_t)].

h_RT (Output) Array of non-dimensional pure species enthalpies [double precision array]. Must be dimensioned at least as large as the number of species in *mix*.

64) getGibbs_RT

Get the array of pure-species non-dimensional Gibbs functions

$$\hat{g}_k(T, P_0)/\hat{R}T, \quad k = 1 \ldots K. \tag{2.7}$$

Syntax

call getGibbs_RT(*mix*, *g_RT*)

Arguments

mix (Input) Mixture object [type(mixture_t)].

g_RT (Output) Array of non-dimensional pure species Gibbs functions [double precision array]. Must be dimensioned at least as large as the number of species in *mix*.

65) getCp_R

Get the array of pure-species non-dimensional heat capacities at constant pressure

$$\hat{c}_{p,k}(T, P_0)/\hat{R}, \quad k = 1 \ldots K. \tag{2.8}$$

Syntax

call getCp_R(*mix*, *cp_R*)

Arguments

mix (Input) Mixture object [type(mixture_t)].

cp_R (Output) Array of non-dimensional pure species heat capacities at constant pressure [double precision array]. Must be dimensioned at least as large as the number of species in *mix*.

66) getEntropy_R

Get the array of pure-species non-dimensional entropies

$$\hat{s}_k(T, P_0)/\hat{R}, k = 1 \ldots K. \tag{2.9}$$

Syntax

call getEntropy_R(*mix*, *s_R*)

Arguments

mix (Input) Mixture object [type(mixture_t)].

s_R (Output) Array of non-dimensional pure species entropies [double precision array]. Must be dimensioned at least as large as the number of species in *mix*.

2.14 Attributes of the Parameterization

These procedures return attributes of the thermodynamic property parameterization used.

67) minTemp

Lowest temperature for which the thermodynamic properties are valid [K].

Syntax

result = minTemp(*mix*)

Arguments

mix (Input) Mixture object [type(mixture_t)].

Result

Returns a `double precision` value.

68) maxTemp

Highest temperature for which the thermodynamic properties are valid [K].

Syntax

result = maxTemp(*mix*)

Arguments

mix (Input) Mixture object [type(mixture_t)].

Result

Returns a `double precision` value.

69) refPressure

The reference (standard-state) pressure P_0. Species thermodynamic property data are specified as a function of temperature at pressure P_0. Usually, P_0 is 1 atm, but some property data sources use 1 bar. Any choice is acceptable, as long as the data for all species in the mixture use the same one.

Syntax

result = refPressure(*mix*)

Arguments

mix (Input / Output) Mixture object [type(mixture_t)].

Result

Returns a `double precision` value.

2.15 Mixture Thermodynamic Properties

2.15.1 Temperature, Pressure, and Density

70) temperature

The temperature [K].

Syntax

result = temperature(*mix*)

Arguments

mix (Input) Mixture object [type(mixture_t)].

Result

Returns a `double precision` value.

71) density

The density [kg/m^3].

Syntax

result = density(*mix*)

Arguments

mix (Input) Mixture object [type(mixture_t)].

Result

Returns a `double precision` value.

72) molarDensity

The number of moles per unit volume [kmol/m^3].

Syntax

result = molarDensity(*mix*)

Arguments

mix (Input) Mixture object [type(mixture_t)].

Result

Returns a `double precision` value.

73) pressure

The pressure [Pa].

Syntax

result = pressure(*mix*)

Arguments

mix (Input) Mixture object [type(mixture_t)].

Result

Returns a `double precision` value.

2.15.2 Molar Mixture Properties

The procedures in this section all return molar thermodynamic properties. The names all end in _*mole*, which should be read as "per mole."

74) enthalpy_mole

Molar enthalpy [J/kmol]

Syntax

result = enthalpy_mole(*mix*)

Arguments

mix (Input / Output) Mixture object [type(mixture_t)].

Result

Returns a `double precision` value.

75) intEnergy_mole

Molar internal energy [J/kmol]

Syntax

result = intEnergy_mole(*mix*)

Arguments

mix (Input / Output) Mixture object [type(mixture_t)].

Result
Returns a `double precision` value.

76) entropy_mole

Molar entropy [J/kmol/K]

Syntax
result = entropy_mole(*mix*)

Arguments

mix (Input / Output) Mixture object [type(mixture_t)].

Result
Returns a `double precision` value.

77) gibbs_mole

Molar Gibbs function [J/kmol/K]

Syntax
result = gibbs_mole(*mix*)

Arguments

mix (Input / Output) Mixture object [type(mixture_t)].

Result
Returns a `double precision` value.

78) cp_mole

Molar heat capacity at constant pressure [J/kmol/K]

Syntax
result = cp_mole(*mix*)

Arguments

mix (Input) Mixture object [type(mixture_t)].

Result
Returns a `double precision` value.

79) cv_mole

Molar heat capacity at constant volume [J/kmol/K]

Syntax

result = cv_mole(*mix*)

Arguments

mix (Input) Mixture object [type(mixture_t)].

Result

Returns a `double precision` value.

80) getChemPotentials_RT

Get the species chemical potentials [J/kmol].

Syntax

call getChemPotentials_RT(*mix*, *mu*)

Arguments

mix (Input) Mixture object [type(mixture_t)].

mu (Output) Species chemical potentials [double precision array]. Must be dimensioned at least as large as the number of species in *mix*.

2.15.3 Specific Mixture Properties

The procedures in this section all return thermodynamic properties per unit mass. The names all end in *_mass*, which should be read as "per (unit) mass."

81) enthalpy_mass

Specific enthalpy [J/kg]

Syntax

result = enthalpy_mass(*mix*)

Arguments

mix (Input) Mixture object [type(mixture_t)].

Result

Returns a `double precision` value.

82) intEnergy_mass

Specific internal energy [J/kg]

Syntax

result = intEnergy_mass(*mix*)

Arguments

mix (Input) Mixture object [type(mixture_t)].

Result

Returns a `double precision` value.

83) entropy_mass

Specific entropy [J/kg/K]

Syntax

result = entropy_mass(*mix*)

Arguments

mix (Input) Mixture object [type(mixture_t)].

Result

Returns a `double precision` value.

84) gibbs_mass

Specific Gibbs function [J/kg]

Syntax

result = gibbs_mass(*mix*)

Arguments

mix (Input) Mixture object [type(mixture_t)].

Result

Returns a `double precision` value.

85) cp_mass

Specific heat at constant pressure [J/kg/K]

Syntax

result = cp_mass(*mix*)

Arguments

mix (Input) Mixture object [type(mixture_t)].

Result

Returns a `double precision` value.

86) cv_mass

Specific heat at constant volume [J/kg/K]

Syntax

result = cv_mass(*mix*)

Arguments

mix (Input) Mixture object [type(mixture_t)].

Result

Returns a `double precision` value.

2.16 Potential Energy

For problems in which external conservative fields play an important role, it may be desirable to assign potential energies to the species. The potential energy adds to the internal energy, but does not affect the entropy. Species-specific potential energies (for example, proportional to charge) can alter the reaction equilibrium constants, and therefore both kinetics and the chemical equilibrium composition. These effects are important in electrochemistry

87) setPotentialEnergy

Set the potential energy of species k [J/kmol] due to external conservative fields (electric, gravitational, or other). Default: 0.0.

Syntax

call setPotentialEnergy(*mix*, *k*, *pe*)

Arguments

mix	(Input / Output) Mixture object [type(mixture_t)].
k	(Input) Species index [integer].
pe	(Input) Potential energy [double precision].

88) potentialEnergy

The potential energy of species k.

Syntax

result = potentialEnergy(*mix*, *k*)

Arguments

mix	(Input / Output) Mixture object [type(mixture_t)].
k	(Input) Species index [integer].

Result

Returns a `double precision` value.

2.17 Critical State Properties

These procedures return parameters of the critical state. Not all equation of state managers implement these, in which case the value **`Undefined`** is returned.

89) critTemperature

The critical temperature [K].

Syntax

result = critTemperature(*mix*)

Arguments

mix	(Input) Mixture object [type(mixture_t)].

Result

Returns a `double precision` value.

90) critPressure

The critical pressure [Pa].

Syntax

result = critPressure(*mix*)

Arguments

mix (Input) Mixture object [type(mixture_t)].

Result

Returns a `double precision` value.

Example

```
use cantera
type(mixture_t) h2o
h2o = Water()
write(*,*) 'Pcrit = ',critPressure(h2o)
```

Output:

```
Pcrit = 22089000.0000000
```

2.18 Saturation Properties

These procedures are only implemented by models that treat saturated liquid/vapor states. Calling them with any other equation of state manager installed will result in an error.

91) satTemperature

The saturation temperature at pressure *p* [K]. An error results if $P > P_{crit}$.

Syntax

result = satTemperature(*mix*, *p*)

Arguments

mix (Input) Mixture object [type(mixture_t)].

p (Input) Pressure [Pa] [double precision].

Result

Returns a `double precision` value.

Example

```
use cantera
type(mixture_t) h2o
h2o = Water()
write(*,*) 'Tsat @ 1 atm = ',satTemperature(h2o,OneAtm)
```

Output:

```
 Tsat @ 1 atm =    373.177232956890
```

92) satPressure

The saturation pressure (vapor pressure) at temperature *t* [K]. An error results if $T > T_{crit}$.

Syntax

result = satPressure(*mix*, *t*)

Arguments

mix	(Input) Mixture object [type(mixture_t)].
t	(Input) Temperature [K] [double precision].

Result

Returns a `double precision` value.

Example

```
use cantera
type(mixture_t) h2o
h2o = Water()
write(*,*) 'Psat @ 300 K = ',satPressure(h2o,300.d0)
```

Output:

```
 Psat @ 300 K =    3528.21380534104
```

2.19 Equations of State

93) equationOfState

Return the equation of state object.

Syntax

result = equationOfState(*mix*)

Arguments

mix (Input / Output) Mixture object [type(mixture_t)].

Result

Returns a `type(eos_t)` value.

94) setEquationOfState

Set the equation of state.

Syntax

call setEquationOfState(*mix, eos*)

Arguments

mix (Input / Output) Mixture object [type(mixture_t)].

eos (Input / Output) Equation of state object [type(eos_t)].

CHAPTER

THREE

Chemical Equilibrium

A discussion of chemical equilibrium, the solver algorithm, etc. will soon replace this box...

95) equilibrate

Chemically equilibrate a mixture.

Syntax

call equilibrate(*mix*, *propPair*)

Arguments

mix (Input / Output) Mixture object [type(mixture_t)].

propPair (Input) Property pair to hold fixed [character*(*)].

Description

This subroutine sets the mixture to a state of chemical equilibrium, holding two thermodynamic properties fixed at their initial values. The pair of properties is specified by a two-character string, as shown in the table below.

String	Property 1	Property 2
TP	temperature	pressure
TV	temperature	specific volume
HP	specific enthalpy	pressure
SP	specific entropy	pressure
SV	specific entropy	specific volume
UV	specific internal energy	specific volume

Example

```
use cantera
type(mixture_t) mix
mix = GRIMech30()
call setState_TPX(mix, 300.d0, OneAtm,
```

```
$     'CH4:0.9, O2:2, N2:7.52')
 call equilibrate(mix,'HP')
 call printSummary(mix, 6)
```

Output:

```
      temperature         2134.24  K
         pressure          101325  Pa
          density        0.158033  kg/m^3
 mean mol. weight         27.6748  amu

                        1 kg            1 kmol
                     -----------     ------------
        enthalpy         -230208      -6.371e+006     J
 internal energy         -871370      -2.412e+007     J
         entropy          9727.6       2.692e+005     J/K
   Gibbs function   -2.09912e+007      -5.809e+008     J
heat capacity c_p         1484.63       4.109e+004     J/K
heat capacity c_v         1184.22       3.277e+004     J/K

                           X               Y
                     -------------     ------------
                H2   9.339031e-004     6.802905e-005
                 H   1.174265e-004     4.276898e-006
                 O   2.387901e-004     1.380495e-004
                O2   1.847396e-002     2.136037e-002
                OH   2.687186e-003     1.651391e-003
               H2O   1.699745e-001     1.106474e-001
               HO2   9.949030e-007     1.186585e-006
              H2O2   6.557154e-008     8.059304e-008
                 C   4.158314e-019     1.804749e-019
                CH   4.275343e-020     2.011257e-020
               CH2   1.037590e-019     5.259063e-020
            CH2(S)   0.000000e+000     0.000000e+000
               CH3   5.912788e-019     3.212273e-019
               CH4   2.440321e-019     1.414648e-019
                CO   2.327608e-003     2.355844e-003
               CO2   8.382804e-002     1.333077e-001
               HCO   7.389023e-011     7.747780e-011
              CH2O   8.885994e-013     9.641078e-013
             CH2OH   1.132705e-018     1.270211e-018
              CH3O   1.686218e-020     1.890919e-020
             CH3OH   7.627499e-020     8.831259e-020
               C2H   0.000000e+000     0.000000e+000
              C2H2   0.000000e+000     0.000000e+000
              C2H3   0.000000e+000     0.000000e+000
              C2H4   0.000000e+000     0.000000e+000
              C2H5   0.000000e+000     0.000000e+000
```

C2H6	0.000000e+000	0.000000e+000
HCCO	0.000000e+000	0.000000e+000
CH2CO	0.000000e+000	0.000000e+000
HCCOH	0.000000e+000	0.000000e+000
N	4.723857e-009	2.390824e-009
NH	5.297876e-010	2.874303e-010
NH2	1.590979e-010	9.211144e-011
NH3	4.043624e-010	2.488376e-010
NNH	2.194905e-010	2.301699e-010
NO	3.077021e-003	3.336222e-003
NO2	1.281909e-006	2.130992e-006
N2O	1.649231e-007	2.622863e-007
HNO	2.693730e-008	3.018755e-008
CN	2.467159e-015	2.319442e-015
HCN	7.246531e-013	7.076590e-013
H2CN	8.061030e-020	8.165586e-020
HCNN	0.000000e+000	0.000000e+000
HCNO	6.903304e-018	1.073236e-017
HOCN	1.190410e-013	1.850695e-013
HNCO	5.138356e-011	7.988446e-011
NCO	2.247139e-012	3.411713e-012
N2	7.183390e-001	7.271270e-001
AR	0.000000e+000	0.000000e+000
C3H7	0.000000e+000	0.000000e+000
C3H8	0.000000e+000	0.000000e+000
CH2CHO	0.000000e+000	0.000000e+000
CH3CHO	0.000000e+000	0.000000e+000

CHAPTER

FOUR

Homogeneous Kinetics

A discussion of homogeneous kinetics models, kinetics managers, etc. will soon replace this box...

4.1 Procedures

96) getReactionString

Reaction equation string.

Syntax

call getReactionString(*mix*, *i*, *rxnString*)

Arguments

mix	(Input) Mixture object [type(mixture_t)].
i	(Input) Reaction index [integer].
rxnString	(Output) Reaction equation string [character*(*)].

Description

This procedure generates the reaction equation as a character string. The notation of Kee et al. [1989] is used. In particular,

1. The reactants and products are separated by `'<=>'` for reversible reactions, and by `'=>'` for irreversible ones;
2. For pressure-dependent falloff reactions in which any species can act as a third-body collision partner, the participation of the third body is denoted by `'(+ M)'`. If only one species acts as the third body, the species name replaces `M`, for example, `'(+ H2O)'`.
3. For reactions involving a third body but which are always in the low-pressure limit (rate proportional to third body concentration), the participation of the third body is denote by `'+ M'`.

Example

```
      use cantera
      character(30) str
      type(mixture_t) mix
      mix = GRIMech30()
      do i = 1,20
         call getReactionString(mix,i,str)
         write(*,10) i,str
10       format(i2,') ',a)
      end do
      return
```

Output:

```
 1) 2 O + M <=> O2 + M
 2) O + H + M <=> OH + M
 3) O + H2 <=> H + OH
 4) O + HO2 <=> OH + O2
 5) O + H2O2 <=> OH + HO2
 6) O + CH <=> H + CO
 7) O + CH2 <=> H + HCO
 8) O + CH2(S) <=> H2 + CO
 9) O + CH2(S) <=> H + HCO
10) O + CH3 <=> H + CH2O
11) O + CH4 <=> OH + CH3
12) O + CO (+ M) <=> CO2 (+ M)
13) O + HCO <=> OH + CO
14) O + HCO <=> H + CO2
15) O + CH2O <=> OH + HCO
16) O + CH2OH <=> OH + CH2O
17) O + CH3O <=> OH + CH2O
18) O + CH3OH <=> OH + CH2OH
19) O + CH3OH <=> OH + CH3O
20) O + C2H <=> CH + CO
```

97) reactantStoichCoeff

The stoichiometric coefficient for species *k* as a reactant in reaction *i*.

Syntax

result = reactantStoichCoeff(*mix*, *k*, *i*)

Arguments

mix (Input) Mixture object [type(mixture_t)].

k (Input) Species index [integer].

i (Input) Reaction index [integer].

Result

Returns a `double precision` value.

Description

A given species may participate in a reaction as a reactant, a product, or both. This function returns the number of molecules of the k^{th} species appearing on the reactant side of the reaction equation. It does not include any species k molecules that act as third-body collision partners but are not written in the reaction equation. Non-integral values are allowed.

Example

```
use cantera
real(8) rcoeff
type(mixture_t) mix
mix = GRIMech30()
inh = speciesIndex(mix,'NH')
write(*,*) 'Reactions in GRI-Mech 3.0 with NH as a reactant:'
do i = 1,nReactions(mix)
   rcoeff = reactantStoichCoeff(mix,inh,i)
   if (rcoeff .gt. 0.d0) write(*,*) '   reaction ',i
end do
```

Output:

```
 Reactions in GRI-Mech 3.0 with NH as a reactant:
    reaction          190
    reaction          191
    reaction          192
    reaction          193
    reaction          194
    reaction          195
    reaction          196
    reaction          197
    reaction          198
    reaction          199
    reaction          280
```

See Also

productStoichCoeff

98) productStoichCoeff

The stoichiometric coefficient for species *k* as a product in reaction *i*.

Syntax

result = productStoichCoeff(*mix*, *k*, *i*)

Arguments

mix	(Input) Mixture object [type(mixture_t)].
k	(Input) Species index [integer].
i	(Input) Reaction index [integer].

Result

Returns a `double precision` value.

Description

A given species may participate in a reaction as a reactant, a product, or both. This function returns the number of molecules of the k^{th} species appearing on the product side of the reaction equation. It does not include any species k molecules that act as third-body collision partners but are not written in the reaction equation. Non-integral values are allowed.

Example

```
use cantera
real(8) pcoeff
type(mixture_t) mix
mix = GRIMech30()
inh = speciesIndex(mix,'NH')
write(*,*) 'Reactions in GRI-Mech 3.0 with NH as a product:'
do i = 1,nReactions(mix)
   pcoeff = productStoichCoeff(mix,inh,i)
   if (pcoeff .gt. 0.d0) write(*,*) '   reaction ',i
end do
```

Output:

```
 Reactions in GRI-Mech 3.0 with NH as a product:
    reaction          200
    reaction          202
    reaction          203
    reaction          208
    reaction          223
    reaction          232
    reaction          242
    reaction          243
    reaction          262
    reaction          269
```

99) netStoichCoeff

The net stoichiometric coefficient for species *k* in reaction *i*.

Syntax

result = netStoichCoeff(*mix*, *k*, *i*)

Arguments

mix	(Input) Mixture object [type(mixture_t)].
k	(Input) Species index [integer].
i	(Input) Reaction index [integer].

Result

Returns a `double precision` value.

Description

This function returns the difference between the product and reactant stoichiometric coefficients.

100) getFwdRatesOfProgress

Get the forward rates of progress $Q_i^{(f)}$ for the reactions [kmol/m^3/s].

Syntax

call getFwdRatesOfProgress(*mix*, *fwdrop*)

Arguments

mix	(Input) Mixture object [type(mixture_t)].
fwdrop	(Output) Forward rates of progress [double precision array]. Must be dimensioned at least as large as the number of reactions in *mix*.

Description

The expressions for the reaction rates of progress are determined by the kinetics model used. See section **??**.

Example see the example for procedure **??**.

101) getRevRatesOfProgress

Get the reverse rates of progress $Q_i^{(r)}$ for the reactions [kmol/m^3/s].

Syntax

call getRevRatesOfProgress(*mix*, *revrop*)

Arguments

mix	(Input) Mixture object [type(mixture_t)].

revrop (Output) Reverse rates of progress [double precision array]. Must be dimensioned at least as large as the number of reactions in *mix*.

Description

The reverse rate of progress is non-zero only for reversible reactions. The expressions for the reaction rates of progress are determined by the kinetics model used. See section **??**.

Example see the example for procedure **??**.

102) getNetRatesOfProgress

Get the net rates of progress for the reactions [kmol/m^3/s].

Syntax

call getNetRatesOfProgress(*mix*, *netrop*)

Arguments

mix (Input) Mixture object [type(mixture_t)].

netrop (Output) Net rates of progress [double precision array]. Must be dimensioned at least as large as the number of reactions in *mix*.

Description

The net rates of progress are the difference between the forward and the reverse rates:

$$Q_i = Q_i^{(f)} - Q_i^{(r)} \tag{4.1}$$

The expressions for the rates of progress are determined by the kinetics model used. See section **??**.

In the example below, a mixture is first created containing hydrogen, oxygen, and argon, and then set to the chemical equilibrium state at 2500 K and 1 atm. The temperature is then lowered, and the forward, reverse, and net rates of progress are computed for all reactions, and those above a threshold are printed. (If the temperature had not been changed, then the forward and reverse rates of progress would be equal for all reversible reactions).getFwdRatesOfProgress, getRevRatesOfProgress

Example

```
use cantera
real(8)  frop(500), rrop(500), netrop(500)
type(mixture_t) mix
character*40 rxn
mix = GRIMech30()
call setState_TPX(mix,2500.d0,OneAtm,'H2:2,O2:1.5,AR:8')
call equilibrate(mix,'TP')
call setState_TP(mix, 2000.d0,OneAtm)
call getFwdRatesOfProgress(mix,frop)
call getRevRatesOfProgress(mix,rrop)
call getNetRatesOfProgress(mix,netrop)
```

```
      do i = 1,nReactions(mix)
         if (abs(frop(i)) .gt. 1.d-10 .or.
     $        abs(rrop(i)) .gt. 1.d-10) then
            call getReactionString(mix,i,rxn)
            write(*,20) rxn,frop(i),rrop(i),netrop(i)
 20         format(a,3e14.5)
         end if
      end do
```

Output:

```
2 O + M <=> O2 + M                0.44094E-03   0.94026E-06   0.44000E-03
O + H + M <=> OH + M              0.51156E-03   0.23776E-05   0.50919E-03
O + H2 <=> H + OH                 0.36968E+01   0.41079E+01  -0.41109E+00
O + HO2 <=> OH + O2               0.13797E-01   0.89268E-03   0.12904E-01
O + H2O2 <=> OH + HO2             0.42931E-03   0.18608E-03   0.24323E-03
H + O2 + M <=> HO2 + M            0.16806E-02   0.12072E-03   0.15599E-02
H + 2 O2 <=> HO2 + O2             0.13767E-02   0.98895E-04   0.12779E-02
H + O2 + H2O <=> HO2 + H2O        0.11154E+00   0.80119E-02   0.10352E+00
H + O2 + AR <=> HO2 + AR          0.21897E-01   0.15729E-02   0.20324E-01
H + O2 <=> O + OH                 0.66073E+01   0.14401E+02  -0.77937E+01
2 H + M <=> H2 + M                0.19043E-03   0.79650E-06   0.18964E-03
2 H + H2 <=> 2 H2                 0.32781E-05   0.13711E-07   0.32644E-05
2 H + H2O <=> H2 + H2O            0.55931E-03   0.23394E-05   0.55697E-03
H + OH + M <=> H2O + M            0.28712E-01   0.57896E-04   0.28654E-01
H + HO2 <=> O + H2O               0.13263E-02   0.81148E-04   0.12452E-02
H + HO2 <=> O2 + H2               0.13544E-01   0.78862E-03   0.12755E-01
H + HO2 <=> 2 OH                  0.28319E-01   0.39935E-02   0.24325E-01
H + H2O2 <=> HO2 + H2             0.22868E-03   0.89198E-04   0.13948E-03
H + H2O2 <=> OH + H2O             0.70669E-04   0.18741E-05   0.68795E-04
OH + H2 <=> H + H2O               0.21063E+02   0.10155E+02   0.10908E+02
2 OH (+ M) <=> H2O2 (+ M)         0.15268E+00   0.11609E-01   0.14107E+00
2 OH <=> O + H2O                  0.32276E+02   0.14003E+02   0.18273E+02
OH + HO2 <=> O2 + H2O             0.48157E-01   0.13518E-02   0.46806E-01
OH + H2O2 <=> HO2 + H2O           0.23253E-03   0.43728E-04   0.18881E-03
OH + H2O2 <=> HO2 + H2O           0.13444E+00   0.25281E-01   0.10916E+00
2 HO2 <=> O2 + H2O2               0.26438E-06   0.39466E-07   0.22492E-06
2 HO2 <=> O2 + H2O2               0.27669E-04   0.41303E-05   0.23539E-04
OH + HO2 <=> O2 + H2O             0.18695E+00   0.52480E-02   0.18170E+00
```

103) getEquilibriumConstants

Get the equilibrium constants of the reactions in concentration units [(kmol/m^3)$^{\Delta n}$, where Δn is the net change in mole numbers in going from reactamts to products].

Syntax

call getEquilibriumConstants(*mix, kc*)

Arguments

mix (Input) Mixture object [type(mixture_t)].

kc (Output) Equilibrium constants [double precision array]. Must be dimensioned at least as large as the number of species in *mix*.

104) getCreationRates

Get the species chemical creation rates [kmol/m^3/s].

Syntax

call getCreationRates(*mix, cdot*)

Arguments

mix (Input) Mixture object [type(mixture_t)].

cdot (Output) Creation rates [double precision array]. Must be dimensioned at least as large as the number of species in *mix*.

Description

The creation rate of species k is

$$\dot{C}_k = \sum_i \left(\nu_{ki}^{(p)} Q_{f,i} + \nu_{ki}^{(r)} Q_{r,i} \right) \tag{4.2}$$

Example see the example for procedure 106.

105) getDestructionRates

Get the species chemical destruction rates [kmol/m^3/s].

Syntax

call getDestructionRates(*mix, ddot*)

Arguments

mix (Input) Mixture object [type(mixture_t)].

ddot (Output) Destruction rates [double precision array]. Must be dimensioned at least as large as the number of species in *mix*.

Description

The destruction rate of species k is

$$\dot{D}_k = \sum_i \left(\nu_{ki}^{(r)} Q_{f,i} + \nu_{ki}^{(p)} Q_{r,i} \right) \tag{4.3}$$

Example see the example for procedure 106.

106) getNetProductionRates

Get the species net chemical production rates [kmol/m^3/s].

Syntax

call getNetProductionRates(*mix, wdot*)

Arguments

mix (Input) Mixture object [type(mixture_t)].

wdot (Output) Net production rates [double precision array]. Must be dimensioned at least as large as the number of species in *mix*.

Description

This procedure returns the net production rates for all species, which is the difference between the species creation and destruction rates:

$$\dot{\omega}_k = \dot{C}_k - \dot{D}_k, \tag{4.4}$$

$$= \sum_i \nu_{ki} Q_i \tag{4.5}$$

If only the net rates are required, it is usually more efficient to call this procedure than to first compute the creation and destruction rates separately and take the difference.

In the example below, the creation, destruction, and net production rates are computed for the same conditions as in the example for procedure **??**. Also shown is the characteristic chemical time, defined as the concentration divided by the destruction rate.

Example

```
use cantera
real(8)  cdot(100), ddot(100), wdot(100), conc(100)
type(mixture_t) mix
character*10 names(100)
mix = GRIMech30()
call setState_TPX(mix,2500.d0,OneAtm,'H2:2,O2:1.5,AR:8')
call equilibrate(mix,'TP')
call setState_TP(mix, 2000.d0,OneAtm)
call getCreationRates(mix,cdot)
call getDestructionRates(mix,ddot)
```

```
      call getNetProductionRates(mix,wdot)
      call getSpeciesNames(mix, names)
      call getConcentrations(mix, conc)
      do k = 1,nSpecies(mix)
         if (wdot(k).gt.1.d-10) then
            write(*,20) names(k), cdot(k), ddot(k), wdot(k),
     $           conc(k)/ddot(k)
 20         format(a,4e14.5)
         end if
      end do
```

Output:

```
H             0.39176E+02   0.21081E+02   0.18095E+02   0.51077E-06
O             0.42994E+02   0.32117E+02   0.10877E+02   0.58471E-06
O2            0.14675E+02   0.67536E+01   0.79217E+01   0.40874E-04
H2O           0.53859E+02   0.24310E+02   0.29549E+02   0.44237E-04
H2O2          0.17831E+00   0.14701E+00   0.31294E-01   0.11045E-07
```

Example 4.1

In the example below, getNetProductionRates is used to evaluate the right-hand side of the species equation for a constant-volume kinetics simulation.

$$dY_k/dt = \dot{\omega}_k M_k/\rho.$$

```
type (mixture_t) :: mix
parameter (KMAX = 100)
double precision wdot(KMAX)
double precision wt(KMAX)
...
call getNetProductionRates(mix, wdot)
call getMolecularWeights(mix, wt)
rho = density(mix)
do k = 1, nSpecies(mix)
   ydot(k) = wdot(k)*wt(k)/rho
end do
```

CHAPTER

FIVE

Transport Properties

A discussion of the theory of transport properties and the transport models implemented in Cantera will soon replace this box...

5.1 Derived Types

transport_t
Derived type for transport managers.

hndl integer . Handle encoding pointer to the C++ object

iok integer . Internal flag used to signify whether or not the object has been properly constructed.

5.2 Procedures

107) MultiTransport

Construct a new multicomponent transport manager.

Syntax
object = MultiTransport(*mix*, *database*, *logLevel*)

Arguments

mix (Input / Output) Mixture object whose transport properties the transport manager will compute. [type(mixture_t)].

database (Input) Transport database file. [character*(*)].

logLevel (Input) A log file in XML format 'transport_log.xml' is generated during construction of the transport manager, containing polynomial fit coefficients, etc. The value of *logLevel* (0 - 4) determines how much detail to write to the log file. [integer].

Result

Returns an object of type **`transport_t`**.

Description

This constructor creates a new transport manager that implements a multicomponent transport model for ideal gas mixtures. The model is based on that of **?** and Kee et al. [1986]. See Section ** for more details.

Example

```
type(mixture_t) mix
type(transport_t) tr
mix = GRIMech30()
tr = MultiTransport(mix, 'gri30_tran.dat', 0)
```

108) MixTransport

Construct a new transport manager that uses a mixture-averaged formulation.

Syntax

object = MixTransport(*mix*, *database*, *logLevel*)

Arguments

mix (Input / Output) Mixture object whose transport properties the transport manager will compute. [type(mixture_t)].

database (Input) Transport database file. [character*(*)].

logLevel (Input) A log file in XML format 'transport_log.xml' is generated during construction of the transport manager, containing polynomial fit coefficients, etc. The value of *logLevel* (0 - 4) determines how much detail to write to the log file. [integer].

Result

Returns an object of type **`transport_t`**.

Description

This constructor creates a new transport manager that implements a 'mixture-averaged' transport model for ideal gas mixtures. The model is very similar to the mixture-averaged model desribed by Kee et al. [1986]. In general, the results of this model are less accurate than those of **`MultiTransport`**, but are far less expensive to compute.

This manager does not implement expressions for thermal diffusion coefficients. If thermal diffusion is important in a problem, use the **`MultiTransport`** transport manager.

Example

```
type(mixture_t) mix
type(transport_t) tr
mix = GRIMech30()
tr = MixTransport(mix, 'gri30_tran.dat', 0)
```

109) setTransport

Set the transport manager to one that was previously created by a call to **`initTransport`**.

Syntax

call setTransport(*mix*, *transportMgr*)

Arguments

mix (Input / Output) Mixture object [type(mixture_t)].

transportMgr (Input / Output) Transport manager [type(transport_t)].

Description

Transport managers may be swapped in or out of a mixture object. This capability makes it possible, for example, to use a multicomponent transport formulation in flow regions where gradients are large, and switch to a much less expensive model for other regions.

This procedure re-installs a transport manager previously created with **`initTransport`**, and saved with **`transport`**.

Note that transport managers cannot be shared between mixture objects.

Example

```
use cantera
type(mixture_t) mix
type(transport_t) tr1, tr2
mix = GRIMech30()
call initTransport(mix, Multicomponent, 'gri30_tran.dat', 0)
tr1 = transport(mix)
write(*,*) thermalConductivity(mix)
call initTransport(mix, CK_Multicomponent, 'gri30_tran.dat', 0)
tr2 = transport(mix)
write(*,*) thermalConductivity(mix)
call setTransport(mix,tr1)
write(*,*) thermalConductivity(mix)
return
```

Output:

```
0.186896997850329
0.186818124230130
0.186896997850329
```

See Also

transport, initTransport

110) delete

Delete a transport manager

Syntax

call delete(*transportMgr*)

Arguments

transportMgr (Input / Output) Transport manager object [type(transport_t)].

Description

Deleting a transport manager releases the memory associated with its kernel object.

111) transportMgr

Return the transport manager object.

Syntax

result = transportMgr(*mix*)

Arguments

mix (Input) Mixture object [type(mixture_t)].

Result

Returns a `type(transport_t)` value.

Description

This function returns a reference to the currently-installed transport manager object. It may be used to store a transport manager before installing a different one.

Example see the example for procedure **??**.

112) viscosity

The dynamic viscosity [Pa-s].

Syntax

result = viscosity(*mix*)

Arguments

mix (Input) Mixture object [type(mixture_t)].

Result

Returns a `double precision` value.

Description

The stress tensor in a Newtonian fluid is given by the expression

$$\tau_{ij} = -P\delta_{ij} + \mu\left(\frac{\partial v_i}{\partial x_j} + \frac{\partial v_i}{\partial x_j}\right) + \delta_{ij}\lambda_b \text{div}\mathbf{v}. \tag{5.1}$$

where μ is the *viscosity* and λ_b is the *bulk viscosity*. This function returns the value of μ computed by the currently-installed transport manager.

113) getSpeciesViscosities

Get the pure species viscosities [Pa-s].

Syntax

call getSpeciesViscosities(*mix*, *visc*)

Arguments

mix (Input) Mixture object [type(mixture_t)].

visc (Input / Output) Array of pure species viscosities [double precision array]. Must be dimensioned at least as large as the number of species in *mix*.

Description

A common approach to computing the viscosity of a mixture is to use a *mixture rule* to generate a weighted average of the viscosities of the pure species. This procedure returns the viscosities of the pure species that are used in the mixture rule. Transport managers that do not compute viscosity using a mixture rule may not implement this procedure.

114) getSpeciesFluxes

Compute the species net mass fluxes due to diffusion.

Syntax

call getSpeciesFluxes(*mix*, *ndim*, *grad_X*, *grad_T*, *fluxes*)

Arguments

mix (Input) Mixture object [type(mixture_t)].

ndim (Input) Dimensionality (1, 2, or 3) [integer].

grad_X (Input / Output) Array of species mole fraction gradients. The (k,n) entry is the gradient of species k in coordinate direction n. [double precision].

grad_T (Input) Temperature gradient array. By default, thermal diffusion is included, using the temperature gradient specified here. Set to zero to compute the result without thermal diffusion [double precision].

fluxes (Output) Array of species diffusive mass fluxes [double precision].

Description

This procedure returns the diffusive mass fluxes for all species, given the local gradients in mole fraction and temperature. The diffusive mass flux vector for species k is given by

$$\mathbf{j}_k = \rho \mathbf{V}_k Y_k, \tag{5.2}$$

where $\mathbf{V}_k$ is the diffusion velocity of species k. The diffusive mass flux satisfies

$$\sum_k \mathbf{j}_k = 0. \tag{5.3}$$

The inputs to this procedure are a mixture object, which defines the local thermodynamic state, the array of mole fraction *gradients* with elements

$$\text{grad\_X(k,n)} = \left(\frac{\partial X_k}{\partial x_n} \right) \tag{5.4}$$

and the one-dimensional temperature gradient array

$$\text{grad\_T(n)} = \left(\frac{\partial T}{\partial x_n} \right) \ . \tag{5.5}$$

Thermal diffusion is included if any component of the temperature gradient is non-zero.

There are several advantages to calling this procedure rather than evaluating the transport properties and then forming the flux expression in your application program. These are:

Performance. For multicomponent transport models, this procedure uses an algorithm that does not require evaluating the multicomponent diffusion coefficient matrix first, and thereby eliminates the need for matix inversion, resulting in somewhat better performance.

Generality. The expression for the flux in terms of mole fraction and temperature gradients is not the same for all transport models. Using this procedure allows switching easily between multicomponent and Fickian diffusion models.

Example

```
use cantera
implicit double precision (a-h,o-z)
type(mixture_t) mix
character*80 infile, thermofile, xstring, model, trdb
double precision gradx(10,3), gradt(3), fluxes(10,3)
double precision x1(8), x2(8), x3(8)
character*20 name(10)
data x1/8*0.1d0/
```

```
      data x2/0.d0, 0.2d0, 0.d0, 0.2d0, 0.d0, 0.2d0, 0.d0,
     $     0.2d0/
      data x3/0.1d0, 0.2d0, 0.3d0, 0.4d0, 0.d0, 0.d0, 0.d0,
     $     0.d0/

      mix = CKGas('air.inp','therm.dat')
      call getSpeciesNames(mix, name)
      call initTransport(mix, Multicomponent,'gri30_tran.dat', 0)
      nsp = nSpecies(mix)

!  generate dummy gradient data
      do k = 1, nsp
         gradx(k,1) = x1(k) - x2(k)    ! dX(k)/dx1
         gradx(k,2) = x2(k) - x3(k)    ! dX(k)/dx2
         gradx(k,3) = x3(k) - x1(k)    ! dX(k)/dx3
         x1(k) = 0.5*(x1(k) + x2(k))
      end do
      gradt(1) = 1000.d0               ! dT/dx1
      gradt(2) = -500.d0               ! dT/dx2
      gradt(3) = 800.d0                ! dT/dx3

      call setState_TPX(mix, 1800.d0, OneAtm, x1)
      call getSpeciesFluxes(mix, 3, gradx, gradt, fluxes)
      do n = 1,3
         write(*,10) n,n,gradt(n),n
 10      format(//'coordinate direction ',i1,':'
     $        /'dT/dx',i1,' = ',g13.5,
     $        /'species ',15x,' dX/dx',i1,'       mass flux')
         do k = 1,nsp
            write(*,20) name(k),gradx(k,n),fluxes(k,n)
 20         format(a,2e14.5)
         end do
      end do
```

Output:

```
coordinate direction 1:
dT/dx1 =    1000.0
species                 dX/dx1       mass flux
O                     0.10000E+00  -0.72034E-05
O2                   -0.10000E+00   0.88312E-05
N                     0.10000E+00  -0.56038E-05
NO                   -0.10000E+00   0.81268E-05
NO2                   0.10000E+00  -0.14005E-04
N2O                  -0.10000E+00   0.86180E-05
N2                    0.10000E+00  -0.90551E-05
AR                   -0.10000E+00   0.10291E-04
```

```
coordinate direction 2:
dT/dx2 =   -500.00
species                 dX/dx2       mass flux
O                    -0.10000E+00  -0.67305E-05
O2                    0.00000E+00   0.21224E-05
N                    -0.30000E+00   0.18372E-04
NO                   -0.20000E+00   0.19739E-04
NO2                   0.00000E+00   0.32196E-05
N2O                   0.20000E+00  -0.17896E-04
N2                    0.00000E+00   0.18281E-05
AR                    0.20000E+00  -0.20653E-04


coordinate direction 3:
dT/dx3 =    800.00
species                 dX/dx3       mass flux
O                     0.00000E+00   0.15068E-04
O2                    0.10000E+00  -0.10916E-04
N                     0.20000E+00  -0.11766E-04
NO                    0.30000E+00  -0.27733E-04
NO2                  -0.10000E+00   0.97117E-05
N2O                  -0.10000E+00   0.82877E-05
N2                   -0.10000E+00   0.74848E-05
AR                   -0.10000E+00   0.98637E-05
```

115) getMultiDiffCoeffs

Get the multicomponent diffusion coefficients [m^2/s]. This procedure may not be implemented by all transport managers. If not, the *multiDiff* array will be left unchanged, and *ierr* will be set to a negative number.

Syntax

call getMultiDiffCoeffs(*mix*, *ldim*, *multiDiff*)

Arguments

mix (Input) Mixture object [type(mixture_t)].

ldim (Input) Leading dimension of *multiDiff*. Must be at least as large as the number of species in *mix* [integer].

multiDiff (Input) Two-dimensional array of multicomponent diffusion coefficients [double precision array]. Must be dimensioned at least as large as the number of species in *mix*.

116) thermalConductivity

Thermal conductivity [W/m/K].

Syntax

result = thermalConductivity(*mix*)

Arguments

mix (Input) Mixture object [type(mixture_t)].

Result

Returns a `double precision` value.

117) getThermalDiffCoeffs

Get the array of thermal diffusion coefficients.

Syntax

call getThermalDiffCoeffs(*mix, dt*)

Arguments

mix (Input) Mixture object [type(mixture_t)].

dt (Output) Thermal diffusion coefficients [double precision array]. Must be dimensioned at least as large as the number of species in *mix*.

118) setOptions

Set transport options.

Syntax

call setOptions(*linearSolver*, *GMRES_m*, *GMRES_eps*)

Arguments

linearSolver (Input) Linear algebra solver to use. Valid values are 'LU' (LU decomposition), and 'GMRES' (GMRES iterative solver). This setting only affects those transport managers that solve systems of linear equations in order to compute the transport properties. Specifying GMRES may significantly accelerate transport property evaluation in some cases. However, if transport properties are evaluated within functions for which numerical derivatives are computed, then using any iterative method may cause convergence difficulties. In many cases, transport property evaluation can be done outside the function. If not, then LU decomposition should be used. Default: LU decomposition. [character*(*)].

GMRES_m (Input) GMRES 'm' parameter. Number of 'inner' iterations between 'outer' iterations. Default: 100 [integer].

GMRES_eps (Input) GMRES convergence parameter. Default: 10^{-4} [double precision].

Description

This procedure sets various transport options. All arguments but the first one are optional. The recommended way to call this procedure is using keywords, as shown in the example.

Example

```
use cantera
transport_t tr
...
call setOptions(tr, linearSolver = 'GMRES') ! set any one or more
call setOptions(tr, GMRES_eps = 1.e-3, GMRES_m = 50)
```

CHAPTER

SIX

Stirred Reactors

This box will soon be replaced by documentation...

6.1 Reactor Objects

6.1.1 Type `cstr_t`

Objects representing zero-dimensional, stirred reactors are implemented in Fortran 90 by the derived type `cstr_t`, defined below.

cstr_t
Stirred reactors.

hndl	integer . Handle encoding pointer to the C++ object
mix	type(mixture_t) . Mixture object

6.2 Procedures

6.3 Constructors

119) StirredReactor

Construct a stirred reactor for zero-dimensional kinetics simulations.

Syntax
object = StirredReactor()

Result
Returns an object of type **`cstr_t`**.

Description

Objects created using **`StirredReactor`** represent generic stirred reactors. Depending on the components attached or the values set for options, it may represent a closed batch reactor, a reactor with a steady flow through it, or one reactor in a network.

The reactor object internally integrates rate equations for volume, species, and energy in primitive form, with no assumption regarding the equation of state. It uses the CVODE stiff ODE integrator, and can handle typical large reaction mechanisms.

A reactor object can be viewed as a cylinder with a piston. Currently, you can set a time constant of the piston motion. By choosing an appropriate value, you can carry out constant volume simulations, constant pressure ones, or anything in between.

The reactor also has a heat transfer coefficient that can be set. A value of zero (default) results in an adiabatic simulation, while a very large value produces an isothermal simulation.

In C++, more general control of the volume and heat transfer coefficient vs. time is possible, allowing construction of engine simulators, for example. (Surface chemistry can be included also.) This will soon be added to the Fortran interface.

The reactor may have an arbitrary number of inlets and outlets, each of which may be connected to a ”flow device” such as a mass flow controller, a pressure regulator, etc. Additional reactors may be connected to the other end of the flow device, allowing construction of arbitrary reactor networks.

If an object constructed by `StirredReactor()` is used with default options, it will simulate an adiabatic, constant volume reactor with gas-phase chemistry but no surface chemistry.

More documentation coming soon.

Example

```
use cantera
type(cstr_t) reactr
reactr = StirredReactor()
```

120) Reservoir

A reservoir is like a reactor, except that the state never changes after being initialized. No chemistry occurs, and the temperature, pressure, composition, and all other properties of the contents remain at their initial values unless explicitly changed. These are designed to be connected to reactors to provide specified inlet conditions.

Syntax

object = Reservoir()

Result

Returns an object of type **`cstr_t`**.

Example

```
        use cantera
        type(cstr_t) reactr
        type(cstr_t) upstr
        type(flowdev_t) mfc
!       install a reservoir upstream from the reactor,
!       and connect them through a mass flow controller
        reactr = StirredReactor()
        upstr = Reservoir()
        mfc = MassFlowController()
        call install(mfc, upstr, reactr)
```

6.4 Assignment

121) copy

Copy one reactor object to another. The contents of *dest* are overwritten.

Syntax

call copy(*src*, *dest*)

Arguments

src	(Input) Reactor object [type(mixture_t)].
dest	(Output) Reactor object [type(mixture_t)].

Description

This procedure performs a shallow copy – the handle is copied, but not the object it points to.

6.5 Setting Options

These procedures set various optional reactor characteristics.

122) setInitialVolume

Set the initial reactor volume [m^3]. By default, the initial volume is 1.0 m^3.

Syntax

call setInitialVolume(*reac*, *vol*)

Arguments

reac (Input / Output) Reactor object. [type(cstr_t)].

vol (Input) Initial volume [double precision].

Example

```
use cantera
type(cstr_t) r
r = StirredReactor()
call setInitialVolume(r, 3.0)
```

123) setInitialTime

Set the initial time [s]. Default = 0.0 s.

Syntax

call setInitialTime(*reac*, *time*)

Arguments

reac (Input / Output) Reactor object. [type(cstr_t)].

time (Input) Initial time [double precision].

Description

Restarts integration from this time using the current substance state as the initial condition. Note that this routine does no interpolation. It simply sets the clock to the specified time, takes the current state to be the value for that time, and restarts.

Example

```
use cantera
type(cstr_t) r
r = StirredReactor()
call setInitialTime(r, 0.02)
```

124) setMaxStep

Set the maximum step size for integration.

Syntax

call setMaxStep(*reac*, *maxstep*)

Arguments

reac (Input / Output) Reactor object. [type(cstr_t)].

maxstep (Input) Maximum step size [double precision].

Description

Note that on the first call to 'advance,' if the maximum step size has not been set, the maximum step size is set to not exceed the specified end time. This effectively keeps the integrator from integrating out to a large time and then interpolating back to the desired final time. This is done to avoid difficulties when simulating problems with sudden changes, like ignition problems. If used, it must be called before calling **advance** for the first time.

Example

```
use cantera
type(cstr_t) r
r = StirredReactor()
call setMaxStep(r,1.d-3)
```

125) setArea

Set the reactor surface area [m^2]. Can be changed at any time.

Syntax

call setArea(*reac*, *area*)

Arguments

reac (Input / Output) Reactor object [type(cstr_t)].

area (Input) Area [double precision].

Description

Note that this does not update automatically if the volume changes, but may be changed manually at any time. The area affects the total heat loss rate.

Example

```
      use cantera
      type(cstr_t) r
      r = StirredReactor()
! a spherical reactor
      radius = 2.0
      vol = (4.d0/3.d0)*Pi*radius**3
      ar = 4.d0*Pi*radius**2
      call setInitialVolume(r, vol)
      call setArea(r, ar)
```

126) setExtTemp

Set the external temperature T_0 used for heat loss calculations. The heat loss rate is calculated from

$$\dot{Q}_{out} = hA(T - T_0) + \epsilon A(T^4 - T_{0,R}^4). \tag{6.1}$$

See also setArea, setEmissivity, setExtRadTemp. Can be changed at any time.

Syntax

call setExtTemp(*reac*, *ts*)

Arguments

reac	(Input / Output) Reactor object [type(cstr_t)].
ts	(Input) External temperature [double precision].

Example

```
use cantera
type(cstr_t) r
r = StirredReactor()
call setExtTemp(r,500.d0)
```

127) setExtRadTemp

Set the external temperature for radiation. By default, this is the same as the temperature set by setExtTemp. But if setExtRadTemp is called, then subsequent calls to setExtTemp do not modify the value set here. Can be changed at any time. See Eq. (6.1)

Syntax

call setExtRadTemp(*reac*, *trad*)

Arguments

reac	(Input / Output) Reactor object [type(cstr_t)].
trad	(Input) External temperature for radiation [double precision].

Example

```
use cantera
type(cstr_t) jet
jet = StirredReactor()
call setExtTemp(jet,1000.d0)
call setExtRadTemp(jet, 300.d0)
```

128) setHeatTransferCoeff

Set the heat transfer coefficient [W/m^2/K]. Default: 0.d0 See Eq. (6.1). May be changed at any time.

Syntax

call setHeatTransferCoeff(*reac*, *h*)

Arguments

reac (Input / Output) Reactor object [type(cstr_t)].

h (Input) Heat transfer coefficient [double precision].

Example

```
use cantera
type(cstr_t) r
r = StirredReactor()
call setHeatTransferCoeff(r,10.d0)
```

129) setVDotCoeff

The equation used to compute the rate at which the reactor volume changes in response to a pressure difference is

$$\dot{V} = K \left(\frac{P - P_{ext}}{p_{ext}} \right) V_i \tag{6.2}$$

where V_i is the initial volume. The inclusion of V_i in this expression is done only so that K will have units of s^{-1}. May be changed at any time.

Syntax

call setVDotCoeff(*reac*, *k*)

Arguments

reac (Input / Output) Reactor object [type(cstr_t)].

k (Input) Coefficient determining rate at which volume changes in response to a pressure difference [double precision].

Description

The default value is zero, which results in the volume being held constant. To conduct a constant pressure simuulation, set P_{ext} to the initial pressure, and set K faster than characteristic rates for your problem. It is always a good idea to examine the constancy of the pressure in the output.

Example

```
use cantera
type(cstr_t) r
r = StirredReactor()
call setVdotCoeff(r,1.d8)
```

130) setEmissivity

Set the emissivity. May be changed at any time. See Eq. (6.1). May be changed at any time.

Syntax

call setEmissivity(*reac*, *emis*)

Arguments

reac	(Input / Output) Reactor object [type(cstr_t)].
emis	(Input) Emissivity [double precision].

Description

The treatment of radiation here is approximate at best. The emissivity is defined here such that the net radiative power loss from the gas to the walls is $\epsilon A \sigma (T^4 - T_{ext}^4)$. Note that this is the emissivity of the gas, not the walls, and will depend on a characteristic reactor dimension (the mean beam length) in the optically thin limit. In the very optically thick limit, it is unlikely that the radiative loss can be computed with a stirred reactor model, since the wall boundary layers may be optically thick.

Example

```
use cantera
type(cstr_t) r
r = StirredReactor()
call setEmissivity(r,0.001)
```

131) setExtPressure

Set the external pressure [Pa].

Syntax

call setExtPressure(*reac*, *p0*)

Arguments

reac	(Input / Output) Reactor object [type(cstr_t)].
p0	(Input) External pressure [double precision].

Description

This pressure is used to form the pressure difference used to evaluate the rate at which the volume changes.

Example

```
use cantera
type(cstr_t) r
r = StirredReactor()
call setExtPressure(r,OneAtm)
```

6.6 Specifying the Mixture

132) setMixture

Specify the mixture contained in the reactor.

Syntax

call setMixture(*reac*, *mix*)

Arguments

reac	(Input / Output) Reactor object [type(cstr_t)].
mix	(Input / Output) Mixture object [type(mixture_t)].

Description

Note that a pointer to this substance is stored, and as the integration proceeds, the state of the mixture is modified. Nevertheless, one mixture object can be used for multiple reactors. When `advance` is called for each reactor, the mixture state is set to the appropriate state for that reactor.

Example

```
      use cantera
      type(mixture_t) gas
      type(cstr_t) reactr1, reactr2

!  create the gas mixture and set its state
      gas = CKGas('mymech.inp')
      if (.not.ready(mix)) stop
      call setState_TPX(mix, 1200.d0, OneAtm,
     &                  'C2H2:1.0, NH3:2.0, O2:0.5')

!  create a reactor object, and insert the gas
      reactr1 = StirredReactor()
      call setMixture(reactr1, gas)

!  use the same gas object for another reactor
      call setState_TPX(mix, 100.d0, OneAtm,
     &                  'C2H2:1.0, NH3:2.0, O2:0.5')
      reactr2 = StirredReactor()
      call setMixture(reactr2, gas)
```

133) contents

Get the mixture contained in the reactor.

Syntax

result = contents(*reac*)

Arguments

reac (Input / Output) Reactor object [type(cstr_t)].

Result

Returns a `type(mixture_t)` value.

6.7 Operation

134) advance

Advance the state of the reactor forward in time to *time*.

Syntax

call advance(*reac*, *time*)

Arguments

reac (Input / Output) Reactor object [type(cstr_t)].

time (Input) Final time [s] [double precision].

Description

Note that this method changes the state of the mixture object. On the first call, initialization is carried out and the maximum integrator step size is set. By default, this is set to *time*. To specify a different maximum step size, precede the call to `advance` with a call to `setMaxStep`. Note that this cannot be reset after `advance` has been called.

6.8 Reactor Attributes

135) residenceTime

The residence time [s].

Syntax

result = residenceTime(*reac*)

Arguments

reac (Input / Output) Reactor object. [type(cstr_t)].

Result

Returns a `double precision` value.

136) time

The current time [s].

Syntax

result = time(*reac*)

Arguments

reac (Input / Output) Reactor object. [type(cstr_t)].

Result

Returns a `double precision` value.

137) volume

The reactor volume [m^3].

Syntax

result = volume(*reac*)

Arguments

reac (Input / Output) Reactor object. [type(cstr_t)].

Result

Returns a `double precision` value.

6.9 Mixture Attributes

These procedures return properties of the reactor contents at the time reached by the last call to `advance`. These values are stored in the reactor object, and are not affected by changes to the mixture object after `advance` was last called.

138) density

The density [kg/m^3].

Syntax

result = density(*reac*)

Arguments

reac (Input / Output) Reactor object. [type(cstr_t)].

Result

Returns a `double precision` value.

139) temperature

The temperature [K].

Syntax

result = temperature(*reac*)

Arguments

reac (Input / Output) Reactor object. [type(cstr_t)].

Result

Returns a `double precision` value.

140) enthalpy_mass

The specific enthalpy [J/kg].

Syntax

result = enthalpy_mass(*reac*)

Arguments

reac (Input / Output) Reactor object. [type(cstr_t)].

Result

Returns a `double precision` value.

141) intEnergy_mass

The specific internal energy [J/kg].

Syntax

result = intEnergy_mass(*reac*)

Arguments

reac (Input / Output) Reactor object. [type(cstr_t)].

Result

Returns a `double precision` value.

142) pressure

The pressure [Pa].

Syntax

result = pressure(*reac*)

Arguments

reac (Input / Output) Reactor object. [type(cstr_t)].

Result

Returns a `double precision` value.

143) mass

The total mass ρV.

Syntax

result = mass(*reac*)

Arguments

reac (Input / Output) Reactor object. [type(cstr_t)].

Result

Returns a `double precision` value.

144) enableChemistry

Enable chemistry.

Syntax

call enableChemistry(*reac*)

Arguments

reac (Input / Output) Reactor object. [type(cstr_t)].

145) disableChemistry

Disable chemistry.

Syntax

call disableChemistry(*reac*)

Arguments

reac (Input / Output) Reactor object. [type(cstr_t)].

ct_api ¿

CHAPTER

SEVEN

Flow Devices

"Flow devices" are objects that regulate fluid flow. These objects are designed to be used in conjunction with the stirred

Perhaps these is a better name for these than "flow devices." Any suggestions?

flowdev_t

Object type used to represent flow control devices.

hndl	integer . Handle encoding pointer to the C++ object
upstream	type(cstr_t) . Upstream reactor
downstream	type(cstr_t) . Downstream reactor
type	integer . Flow controller type

7.1 Procedures

7.2 Constructors

These functions construct new flow devices.

146) MassFlowController

Create a new mass flow controller. A mass flow controller maintains a specified mass flow rate, independent of all other conditions.

Syntax

object = MassFlowController()

Result

Returns an object of type **`flowdev_t`**.

147) PressureController

Create a new pressure controller.

Syntax

object = PressureController()

Result

Returns an object of type **`flowdev_t`**.

Description

This device attempts to maintain a specified *upstream* pressure by adjusting its mass flow rate. It is designed to regulate the exhaust flow rate from a reactor, not the inlet flow rate to a reactor (i.e., to decrease the pressure, the valve opens wider) It should be installed on an outlet of the reactor to be controlled. This device will only function correctly if the reactor also has one or more inlets with positive flow rates. It also acts as a check valve, and does not permit reverse flow under any conditions.

7.3 Assignment

148) copy

Perform a shallow copy of one flo device to another. The contents of *dest* are overwritten.

Syntax

call copy(*src*, *dest*)

Arguments

src	(Input) Flow controller object [type(flowdev_t)].
dest	(Output) Flow controller object [type(flowdev_t)].

7.4 Setting Options

149) setSetpoint

Set the setpoint. The units depend on the type of flow control device. May be called at any time.

Syntax

call setSetpoint(*flowDev*, *setpnt*)

Arguments

flowDev (Input / Output) Flow controller object. [type(flowdev_t)].

setpnt (Input) Setpoint. [double precision].

Description

For mass flow controllers, this sets the mass flow rate [kg/s]. For pressure controllers, it sets the pressure setpoint [Pa].

– ¿

150) install

! Install a flow control device between two reactors.

Syntax

call install(*flowDev*, *upstream*, *downstream*)

Arguments

flowDev (Input / Output) Flow controller object. [type(flowdev_t)].

upstream (Input / Output) Upstream reactor object. [type(cstr_t)].

downstream (Input / Output) Downstream reactor object. [type(cstr_t)].

151) upstream

Return a reference to the upstream reactor.

Syntax

result = upstream(*flowDev*)

Arguments

flowDev (Input / Output) Flow controller object. [type(flowdev_t)].

Result

Returns a `type(cstr_t)` value.

152) downstream

Return a reference to the downstream reactor.

Syntax

result = downstream(*flowDev*)

Arguments

flowDev (Input / Output) Flow controller object. [type(flowdev_t)].

Result

Returns a `type(cstr_t)` value.

153) massFlowRate

Mass flow rate [kg/s].

Syntax

result = massFlowRate(*flowDev*)

Arguments

flowDev (Input / Output) Flow controller object. [type(flowdev_t)].

Result

Returns a `double precision` value.

154) update

Update the state of the flow device. This only needs to be called for those flow controllers that have an internal state, such as pressure controllers that use a PID controller.

Syntax

call update(*flowDev*)

Arguments

flowDev (Input / Output) Flow controller object. [type(flowdev_t)].

155) reset

Reset the controller. Call this procedure before operating any flow controller that uses a PID controller.

Syntax

call reset(*flowDev*)

Arguments

flowDev (Input / Output) Flow controller object. [type(flowdev_t)].

156) ready

Returns true if flow controller is ready for use.

Syntax

result = ready(*flowDev*)

Arguments

flowDev (Input / Output) Flow controller object. [type(flowdev_t)].

Result

Returns a `integer` value.

157) setGains

Some flow control devices use a PID (proportional / integral / derivative) controller. This procedure can be used to set the gains for the PID controller.

Syntax

call setGains(*flowDev*, *gains*)

Arguments

flowDev (Input / Output) Flow controller object. [type(flowdev_t)].

gains (Input) gains [double precision array]. Must be dimensioned at least 4.

158) getGains

Some flow control devices use a PID (proportional / integral / derivative) controller. This procedure can be used to retrieve the current gain settings for the PID controller.

Syntax

call getGains(*flowDev, gains*)

Arguments

flowDev	(Input / Output) Flow controller object. [type(flowdev_t)].
gains	(Output) gains [double precision array]. Must be dimensioned at least 4.

159) maxError

Maximum absolute value of the controller error signal (input - setpoint) since the last call to 'reset'.

Syntax

result = maxError(*flowDev*)

Arguments

flowDev	(Input / Output) Flow controller object. [type(flowdev_t)].

Result

Returns a `double precision` value.

ct_api ¿

CHAPTER

EIGHT

Utility Functions

CHAPTER

NINE

Utilities

9.1 Introduction

These procedures carry out various utility functions.

9.2 Procedures

9.3 Procedures

160) printSummary

Write to logical unit *lu* a summary of the mixture state.

Syntax

call printSummary(*mix, lu*)

Arguments

mix	(Input) Mixture object [type(mixture_t)].
lu	(Input) Fortran logical unit for output [integer].

APPENDIX

A

Glossary

substance
A macroscopic sample of matter with a precise, characteristic composition. Pure water is a substance, since it is always made up of H_2O molecules; any pure element is also a substance.

compound
A substance containing more than one element. Sodium chloride, water, and silicon carbide are all compounds.

mixture
A macroscopic sample of matter made by combining two or more substances, usually (but not necessarily) finely-divided and intermixed. Liquid water with dissolved oxygen and nitrogen is a mixture, as are sand, air, wood, beer, and most other everyday materials. In fact, since even highly-purified "substances" contain measurable trace impurities, it is exceptionally rare to encounter anything that is *not* a mixture, i.e., anything that is *truly* a substance.

solution
A mixture in which the constituents are fully mixed on a molecular scale. In a solution, all molecules of a given constituent (or all sites of a given type) are statistically equivalent, and the configurational entropy of the mixture is maximal. Mixtures of gases are solutions, as are mixtures of mutually-soluble liquids or solids. For example, silicon and germanium form a crystalline solid solution Si_xGe_{1-x}, where x is continuously variable over a range of values.

phase
A macroscopic sample of matter with a homogeneous composition and structure that is stable to small perturbations. Example: water at a temperature below its critical temperature and above its triple point can exist in two stable states: a low-density state (vapor) or a high-density one(liquid). There is no stable homogeneous state at intermediate densities, and any attempt to prepare such a state will result in its spontaneous segregation into liquid and vapor regions. Liquid water and water vapor are two phases of water below its critical temperature. (Above it, these two phases merge, and there is only a single phase.)

Note that a phase does not need to be *thermodynamically* (globally) stable. Diamond is a valid phase of carbon at atmospheric pressure, even though graphite is the thermodynamically stable phase.

BIBLIOGRAPHY

R. T. Jacobsen, R. B. Stewart, and A. F. Myers. An equation of state for oxygen and nitrogen. *Adv. Cryo. Engrg.*, 18:248, 1972.

R. J. Kee, G. Dixon-Lewis, J. Warnatz, , M. E. Coltrin, and J. A. Miller. A Fortran computer code package for the evaluation of gas-phase multicomponent transport properties. Technical Report SAND86-8246, Sandia National Laboratories, 1986. An electronic version of this report may be downloaded from http://stokes.lance.colostate.edu/chemkinmanualslist.html.

R. J. Kee, F. M. Rupley, and J. A. Miller. Chemkin-II: A Fortran chemical kinetics package for the analysis of gas-phase chemical kinetics. Technical Report SAND89-8009, Sandia National Laboratories, 1989.

W. C. Reynolds. Thermodynamic properties in si. Technical report, Department of Mechanical Engineering, Stanford University, 1979.

Gregory P. Smith, David M. Golden, Michael Frenklach, Nigel W. Moriarty, Boris Eiteneer, Mikhail Goldenberg, C. Thomas Bowman, Ronald K. Hanson, Soonho Song, William C. Gardiner, Jr., Vitali V. Lissianski, , and Zhiwei Qin. GRI-Mech version 3.0, 1997. see http://www.me.berkeley.edu/gri_mech.

INDEX