

# ExternalMedia Reference Manual

## 1.0

Generated by Doxygen 1.5.4

Thu Jan 24 07:26:19 2008



# Contents

<b>1</b>	<b>External Media HowTo</b>	<b>1</b>
1.1	Introduction . . . . .	1
1.2	Architecture of the package . . . . .	1
1.3	Using an already implemented external medium with Dymola . . . . .	2
1.4	Developing a new external medium model . . . . .	3
<b>2</b>	<b>ExternalMedia Namespace Index</b>	<b>5</b>
2.1	ExternalMedia Namespace List . . . . .	5
<b>3</b>	<b>ExternalMedia Hierarchical Index</b>	<b>7</b>
3.1	ExternalMedia Class Hierarchy . . . . .	7
<b>4</b>	<b>ExternalMedia Class Index</b>	<b>9</b>
4.1	ExternalMedia Class List . . . . .	9
<b>5</b>	<b>ExternalMedia File Index</b>	<b>11</b>
5.1	ExternalMedia File List . . . . .	11
<b>6</b>	<b>ExternalMedia Namespace Documentation</b>	<b>13</b>
6.1	std Namespace Reference . . . . .	13
<b>7</b>	<b>ExternalMedia Class Documentation</b>	<b>15</b>
7.1	BaseSolver Class Reference . . . . .	15
7.2	BaseTwoPhaseMedium Class Reference . . . . .	22
7.3	FluidConstants Struct Reference . . . . .	31
7.4	MediumMap Class Reference . . . . .	32
7.5	SolverMap Class Reference . . . . .	36
7.6	TestSolver Class Reference . . . . .	38
7.7	TwoPhaseMedium Class Reference . . . . .	42
7.8	TwoPhaseMediumProperties Class Reference . . . . .	44

<b>8</b>	<b>ExternalMedia File Documentation</b>	<b>47</b>
8.1	errorhandling.h File Reference . . . . .	47
8.2	externalmedialib.cpp File Reference . . . . .	49
8.3	externalmedialib.h File Reference . . . . .	60
8.4	include.h File Reference . . . . .	71

# Chapter 1

## External Media HowTo

### 1.1 Introduction

Welcome to the ExternalMediaPackage HowTo guide. This package is meant to help you interfacing your favourite external fluid computation code(s) to Modelica, using the Modelica.Media interface. Currently, the ExternalMedia library only supports two-phase media, using Dymola 6 and Microsoft Visual Studio. In the future, we might add support for single-phase pure and mixture media, as well as for OpenModelica and gcc. Please contact the authors if you're interested in such developments.

The ExternalMedia package has been implemented and is maintained by Francesco Casella ([casella@elet.polimi.it](mailto:casella@elet.polimi.it)) and Christoph Richter ([ch.richter@tu-bs.de](mailto:ch.richter@tu-bs.de)). The code is released under the Modelica license.

Copyright (c) 2006, Politecnico di Milano and TU Braunschweig.

### 1.2 Architecture of the package

At the top level there is a Modelica package (ExternalMedia), which contains all the basic infrastructure needed to use external fluid properties computation software through a Modelica.Media compliant interface. In particular, the ExternalMedia.Media.ExternalTwoPhaseMedium package is a full-fledged implementation of a two-phase medium model, compliant with the Modelica.Media.Interfaces.PartialTwoPhaseMedium interface. The ExternalTwoPhaseMedium package can be used with any external fluid property computation software; the specific software to be used is specified by changing the libraryName package constant, which is then handled by the underlying C code to select the appropriate external code to use.

The functions within ExternalTwoPhaseMedium talk to a C/C++ interface layer (called **externalmedialib.cpp** (p. 49)) via external functions calls. This layer takes care of initializing the external packages, and maintains a collection of C++ objects (stored within a **MediumMap** (p. 32) C++ object), corresponding to different instances of the medium model. More specifically, each instance of ExternalTwoPhaseMedium.BaseProperties corresponds to an instance of the C++ object **TwoPhaseMedium** (p. 42), identified by an integer uniqueID; each **TwoPhaseMedium** (p. 42) object also contains a pointer to a C++ object inheriting from **BaseSolver** (p. 15); this is a wrapper object around the actual external code (or solver) which has to be called to compute the fluid properties. It is possible to use multiple instances of possibly different solvers at the same time; the collection of active solver is managed by the **SolverMap** (p. 36) C++ object.

The default implementation of the external medium works as follows. At initialization, each instance of the `BaseProperties` model calls the `createMedium_()` function of the interface layer. This triggers the allocation within the **MediumMap** (p. 32) object of a new **TwoPhaseMedium** (p. 42) object, which will contain all the medium properties, and a pointer to the appropriate solver. This object will be stored in the **MediumMap** (p. 32), and identified with a `uniqueID`, which is returned by the `createMedium()` function and stored within the `BaseProperties` model. The solver object is created by the **SolverMap** (p. 36) object, based on the `mediumName`, `libraryName` and `substanceName` constants of the Modelica medium package

- only one solver object is created for multiple instances of the same external medium.

When one of the `setState` functions within `BaseProperties` is called, the interface layer will call the corresponding `setState` function of the **TwoPhaseMedium** (p. 42) object, which will in turn invoke the `setState` function of the appropriate solver object, to actually compute the fluid properties. These will be stored within the **TwoPhaseMedium** (p. 42) object, for later retrieval. The `setState` function then returns a `ThermodynamicState` record, which only contains integer `uniqueID`, identifying the **TwoPhaseMedium** (p. 42) object in the **MediumMap** (p. 32). Subsequent calls to compute any other functions of that state will just fetch the corresponding values from the **TwoPhaseMedium** (p. 42) object with the given `uniqueID` - the **TwoPhaseMedium** (p. 42) object acts like a cache for already computed properties. The default implementation assumes that all available properties are computed at once by the `setState` functions, so that subsequent calls to any property function will just result in a retrieval from the cache - more sophisticated implementations could be devised, where the `setState` function only computes some properties at first, and flags are provided within the **TwoPhaseMedium** (p. 42) object to keep into account what has already been computed and stored in the cache, and what has still to be computed if need.

With the current implementation, it is not possible to use `setState` functions outside the `BaseProperties` model, unless one explicitly manages and sets the `uniqueID` integers - this is not possible if the user model has to be compatible with other Modelica.Media fluid models, which do not have the `uniqueID`. Support for calls to `setState` functions with automatic management of the `uniqueID` might be added in the future.

### 1.3 Using an already implemented external medium with Dymola

If somebody else has already implemented the external medium, you only need a few things to use it on your computer:

- the `ExternalMedia` Modelica package
- the `ExternalTwoPhaseMedium.lib` (and possibly the `ExternalTwoPhaseMedium.dll`) files, containing the compiled C/C++ code
- the **externalmedialib.h** (p. 60) C header file

Copy the `ExternalTwoPhaseMedium.lib` file within the `Dymola/bin/lib` directory, the `ExternalTwoPhaseMedium.dll` file in the `Windows/System32` directory (only if the library has been compiled to a DLL), and the **externalmedialib.h** (p. 60) file in the `Dymola/Source` directory.

You can check that the compiler setup is working by running the `TestMedium` model in the `ExternalMedia.Test` package - this will simulate a dummy external fluid model, roughly corresponding to liquid water at low pressure and temperature.

You can now use the external media provided in the `ExternalMedia.Media` package. There are two options here: the first is to use `ExternalTwoPhaseMedium` directly, using modifiers to change `libraryName` and `substanceNames` to suit your needs; the other is to use or modify one of the pre-packaged media. The `libraryName` string will be decoded by the **SolverMap** (p. 36) object - for further details on the syntax of the string, ask to the implementor of the interface to the code you'll actually use.

## 1.4 Developing a new external medium model

The `ExternalMedia` package has been designed to ease your task, so that you will only have to write the minimum amount of code which is strictly specific to your external code - everything else is already provided. The following instructions apply if you want to develop an external medium model which include a (sub)set of the functions defined in `Modelica.Media.Interfaces.PartialTwoPhaseMedium`, and you want to compute all the properties at once when you call the `setState` function (this is a reasonable choice, since the majority of the time is usually spent to run preliminary computations (e.g. compute the Helmholtz function), and then computing all the properties only adds a small overhead on top of that time).

First of all, you have to write your own solver object code; you can look at the code of the `TestMedium` and `FluidPropMedium` code as examples. Inherit from the **BaseSolver** (p. 15) object, which provides default implementations for most of the required functions, and then just add your own implementation for the following functions: object constructor, object destructor, `setMediumConstants()`, `setSat_p()`, `setSat_T()`, `setState_ph()`, `setState_pT()`, `setState_ps()`, `setState_dT()`. Note that the `setState` functions need to compute and fill in all the fields of the `properties` record, otherwise the corresponding functions will not work. For example, if your `setState` functions do not set the value of properties, then you won't be able to later call `thermalConductivity(medium.state)` in your Modelica models. On the other hand, you don't necessarily need to implement all of four `setState` functions: if you know in advance that your models will only use certain combinations of variable as inputs (e.g. `p`, `h`), then you might omit implementing the `setState` functions corresponding to the other ones.

Then you must modify the `SolverMap::addSolver()` function, so that it will instantiate your new solver when it is called with the appropriate `libraryName` string. You are free to invent your own syntax for the `libraryName` string, in case you'd like to be able to set up the external medium with some additional configuration data from within Modelica - it's up to you to decode that syntax within the `addSolver()` function, and within the constructor of your solver object.

Finally, add the `.cpp` and `.h` files to the C/C++ project, recompile to a static library (or to a DLL), and then follow the instructions of Section 2. of this document.





## Chapter 2

# ExternalMedia Namespace Index

### 2.1 ExternalMedia Namespace List

Here is a list of all documented namespaces with brief descriptions:

<b>std</b> . . . . .	13
----------------------	----



## Chapter 3

# ExternalMedia Hierarchical Index

### 3.1 ExternalMedia Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

BaseSolver . . . . .	15
TestSolver . . . . .	38
BaseTwoPhaseMedium . . . . .	22
TwoPhaseMedium . . . . .	42
FluidConstants . . . . .	31
MediumMap . . . . .	32
SolverMap . . . . .	36
TwoPhaseMediumProperties . . . . .	44



## Chapter 4

# ExternalMedia Class Index

### 4.1 ExternalMedia Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

<b>BaseSolver</b> (Base solver class ) . . . . .	15
<b>BaseTwoPhaseMedium</b> (Base two-phase medium class ) . . . . .	22
<b>FluidConstants</b> (Fluid constants struct ) . . . . .	31
<b>MediumMap</b> (Medium map ) . . . . .	32
<b>SolverMap</b> (Solver map ) . . . . .	36
<b>TestSolver</b> (Test solver class ) . . . . .	38
<b>TwoPhaseMedium</b> (Two phase medium ) . . . . .	42
<b>TwoPhaseMediumProperties</b> (Two phase medium property class ) . . . . .	44



## Chapter 5

# ExternalMedia File Index

### 5.1 ExternalMedia File List

Here is a list of all documented files with brief descriptions:

<b>errorhandling.h</b> (Error handling for external library ) . . . . .	47
<b>externalmedialib.cpp</b> (Interface layer ) . . . . .	49
<b>externalmedialib.h</b> (Header file to be included in Dymola ) . . . . .	60
<b>include.h</b> (Generic include file ) . . . . .	71





## Chapter 6

# ExternalMedia Namespace Documentation

### 6.1 std Namespace Reference

#### 6.1.1 Detailed Description

Microsoft Visual C++ 2005

This is an example of a client application in Microsoft Visual C++ 2005 for FluidProp, a COM server module for the calculation of fluid properties. FluidProp is a common interface to GasMix, IF97, Refprop, StanMix, TPSI and is developed by Piero Colonna and Teus van der Stelt.

The class defined in this file, TFluidProp, is as a wrapper class for the base class IFluidProp\_COM. TFluidProp hides specific COM server details like safe arrays (SAFEARRAY) and binary strings (BSTR) in IFluidProp\_COM. In the TFluidProp class only standard C++ data types are used. This seriously facilitates working with the COM server.

Teus van der Stelt Energy Technology Section TU Delft

July, 2004, for FluidProp 1 January, 2006, for FluidProp 2 April, 2007, for FluidProp 2.3



## Chapter 7

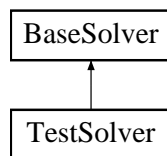
# ExternalMedia Class Documentation

### 7.1 BaseSolver Class Reference

Base solver class.

```
#include <basesolver.h>
```

Inheritance diagram for BaseSolver::



#### Public Member Functions

- **BaseSolver** (const string &mediumName, const string &libraryName, const string &substanceName)  
*Constructor.*
- virtual ~**BaseSolver** ()  
*Destructor.*
- virtual void **setFluidConstants** ()  
*Set fluid constants.*
- virtual void **setSat\_p** (double &p, **TwoPhaseMediumProperties** \*const properties)  
*Set saturation properties from p.*
- virtual void **setSat\_T** (double &T, **TwoPhaseMediumProperties** \*const properties)  
*Set saturation properties from T.*
- virtual void **setSat\_p\_state** (**TwoPhaseMediumProperties** \*const properties)  
*Set saturation properties from within BaseProperties model.*

- virtual void **setState\_dT** (double &d, double &T, int &phase, **TwoPhaseMediumProperties** \*const properties)  
*Set state from d, T, and phase.*
- virtual void **setState\_ph** (double &p, double &h, int &phase, **TwoPhaseMediumProperties** \*const properties)  
*Set state from p, h, and phase.*
- virtual void **setState\_ps** (double &p, double &s, int &phase, **TwoPhaseMediumProperties** \*const properties)  
*Set state from p, s, and phase.*
- virtual void **setState\_pT** (double &p, double &T, **TwoPhaseMediumProperties** \*const properties)  
*Set state from p and T.*
- virtual void **setBubbleState** (int phase, **TwoPhaseMediumProperties** \*const properties, **TwoPhaseMediumProperties** \*const bubbleProperties)  
*Set bubble state.*
- virtual void **setDewState** (int phase, **TwoPhaseMediumProperties** \*const properties, **TwoPhaseMediumProperties** \*const bubbleProperties)  
*Set dew state.*
- virtual bool **computeDerivatives** (**TwoPhaseMediumProperties** \*const properties)  
*Compute derivatives.*
- virtual double **isentropicEnthalpy** (double &p, **TwoPhaseMediumProperties** \*const properties)  
*Compute isentropic enthalpy.*
- double **molarMass** () const  
*Return molar mass (Default implementation provided).*
- double **criticalTemperature** () const  
*Return temperature at critical point (Default implementation provided).*
- double **criticalPressure** () const  
*Return pressure at critical point (Default implementation provided).*
- double **criticalDensity** () const  
*Return density at critical point (Default implementation provided).*
- double **criticalMolarVolume** () const  
*Return molar volume at critical point (Default implementation provided).*
- double **criticalEnthalpy** () const  
*Return specific enthalpy at critical point (Default implementation provided).*
- double **criticalEntropy** () const  
*Return specific entropy at critical point (Default implementation provided).*

## Public Attributes

- string **mediumName**  
*Medium name.*
- string **libraryName**  
*Library name.*
- string **substanceName**  
*Substance name.*

## Protected Attributes

- **FluidConstants** **\_fluidConstants**  
*Fluid constants.*

### 7.1.1 Detailed Description

Base solver class.

This is the base class for all external solver objects (e.g. **TestSolver** (p. 38), **FluidPropSolver**).

Francesco Casella, Christoph Richter, Sep 2006 Copyright Politecnico di Milano and TU Braunschweig

### 7.1.2 Constructor & Destructor Documentation

#### 7.1.2.1 **BaseSolver::BaseSolver** (const string & *mediumName*, const string & *libraryName*, const string & *substanceName*)

Constructor.

The constructor is copying the medium name, library name and substance name to the locally defined variables.

#### Parameters:

*mediumName* Arbitrary medium name  
*libraryName* Name of the external fluid property library  
*substanceName* Substance name

#### 7.1.2.2 **BaseSolver::~~BaseSolver** () [virtual]

Destructor.

The destructor for the base solver if currently not doing anything.

### 7.1.3 Member Function Documentation

#### 7.1.3.1 void BaseSolver::setFluidConstants () [virtual]

Set fluid constants.

This function sets the fluid constants which are defined in the **FluidConstants** (p. 31) record in Modelica. It should be called when a new solver is created.

Must be re-implemented in the specific solver

Reimplemented in **TestSolver** (p. 39).

#### 7.1.3.2 void BaseSolver::setSat\_p (double & *p*, TwoPhaseMediumProperties \*const *properties*) [virtual]

Set saturation properties from *p*.

This function sets the saturation properties for the given pressure *p*. The computed values are written to the two phase medium property struct.

Must be re-implemented in the specific solver

##### Parameters:

*p* Pressure

*properties* Two phase medium property record

Reimplemented in **TestSolver** (p. 39).

#### 7.1.3.3 void BaseSolver::setSat\_T (double & *T*, TwoPhaseMediumProperties \*const *properties*) [virtual]

Set saturation properties from *T*.

This function sets the saturation properties for the given temperature *T*. The computed values are written to the two phase medium property struct.

Must be re-implemented in the specific solver

##### Parameters:

*T* Temperature

*properties* Two phase medium property record

Reimplemented in **TestSolver** (p. 39).

#### 7.1.3.4 void BaseSolver::setSat\_p\_state (TwoPhaseMediumProperties \*const *properties*) [virtual]

Set saturation properties from within BaseProperties model.

This function sets the saturation properties for the given pressure *p* and is designed to be used from within the BaseProperties model in Modelica. The computed values are written to the two phase medium property struct.

Must be re-implemented in the specific solver

**Parameters:**

*properties* Two phase medium property record

Reimplemented in **TestSolver** (p. 40).

### 7.1.3.5 void BaseSolver::setState\_dT (double & *d*, double & *T*, int & *phase*, TwoPhaseMediumProperties \*const *properties*) [virtual]

Set state from *d*, *T*, and *phase*.

This function sets the thermodynamic state record for the given density *d*, the temperature *T* and the specified phase. The computed values are written to the two phase medium property struct.

Must be re-implemented in the specific solver

**Parameters:**

*d* Density

*T* Temperature

*phase* Phase (2 for two-phase, 1 for one-phase, 0 if not known)

*properties* Two phase medium property record

Reimplemented in **TestSolver** (p. 40).

### 7.1.3.6 void BaseSolver::setState\_ph (double & *p*, double & *h*, int & *phase*, TwoPhaseMediumProperties \*const *properties*) [virtual]

Set state from *p*, *h*, and *phase*.

This function sets the thermodynamic state record for the given pressure *p*, the specific enthalpy *h* and the specified phase. The computed values are written to the two phase medium property struct.

Must be re-implemented in the specific solver

**Parameters:**

*p* Pressure

*h* Specific enthalpy

*phase* Phase (2 for two-phase, 1 for one-phase, 0 if not known)

*properties* Two phase medium property record

Reimplemented in **TestSolver** (p. 40).

### 7.1.3.7 void BaseSolver::setState\_ps (double & *p*, double & *s*, int & *phase*, TwoPhaseMediumProperties \*const *properties*) [virtual]

Set state from *p*, *s*, and *phase*.

This function sets the thermodynamic state record for the given pressure *p*, the specific entropy *s* and the specified phase. The computed values are written to the two phase medium property struct.

Must be re-implemented in the specific solver

**Parameters:**

- p* Pressure
- s* Specific entropy
- phase* Phase (2 for two-phase, 1 for one-phase, 0 if not known)
- properties* Two phase medium property record

Reimplemented in **TestSolver** (p. 41).

#### 7.1.3.8 void BaseSolver::setState\_pT (double & *p*, double & *T*, TwoPhaseMediumProperties \*const *properties*) [virtual]

Set state from p and T.

This function sets the thermodynamic state record for the given pressure p and the temperature T. The computed values are written to the two phase medium property struct.

Must be re-implemented in the specific solver

**Parameters:**

- p* Pressure
- T* Temperature
- properties* Two phase medium property record

Reimplemented in **TestSolver** (p. 41).

#### 7.1.3.9 void BaseSolver::setBubbleState (int *phase*, TwoPhaseMediumProperties \*const *properties*, TwoPhaseMediumProperties \*const *bubbleProperties*) [virtual]

Set bubble state.

This function sets the bubble state record bubbleProperties corresponding to the saturation data contained in the properties record.

The default implementation of the setBubbleState function is relying on the correct behaviour of setState\_ph with respect to the state input. Can be overridden in the specific solver code to get more efficient or correct handling of this situation.

**Parameters:**

- phase* Phase (1: one-phase, 2: two-phase)
- properties* Two phase medium property record with saturation properties data
- bubbleProperties* Two phase medium property record where to write the bubble point properties

#### 7.1.3.10 void BaseSolver::setDewState (int *phase*, TwoPhaseMediumProperties \*const *properties*, TwoPhaseMediumProperties \*const *dewProperties*) [virtual]

Set dew state.



This function sets the dew state record `dewProperties` corresponding to the saturation data contained in the `properties` record.

The default implementation of the `setDewState` function is relying on the correct behaviour of `setState_ph` with respect to the state input. Can be overridden in the specific solver code to get more efficient or correct handling of this situation.

**Parameters:**

***phase*** Phase (1: one-phase, 2: two-phase)

***properties*** Two phase medium property record with saturation properties data

***dewProperties*** Two phase medium property record where to write the dew point properties

### 7.1.3.11 `bool BaseSolver::computeDerivatives (TwoPhaseMediumProperties *const properties) [virtual]`

Compute derivatives.

This function computes the derivatives according to the Bridgman's table. The computed values are written to the two phase medium property struct. This function can be called from within the `setState_XX` routines when implementing a new solver. Please be aware that `cp`, `beta` and `kappa` have to be provided to allow the computation of the derivatives. It returns false if the computation failed.

Default implementation provided.

**Parameters:**

***properties*** Two phase medium property record

### 7.1.3.12 `double BaseSolver::isentropicEnthalpy (double & p, TwoPhaseMediumProperties *const properties) [virtual]`

Compute isentropic enthalpy.

This function returns the enthalpy at pressure `p` after an isentropic transformation from the state specified by the `properties` input

Must be re-implemented in the specific solver

**Parameters:**

***p*** New pressure

***properties*** Two phase medium property record corresponding to current state

The documentation for this class was generated from the following files:

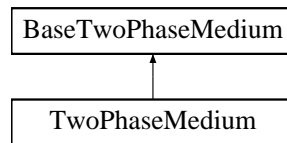
- basesolver.h
- basesolver.cpp

## 7.2 BaseTwoPhaseMedium Class Reference

Base two-phase medium class.

```
#include <basetwophasemedium.h>
```

Inheritance diagram for BaseTwoPhaseMedium::



### Public Member Functions

- **BaseTwoPhaseMedium** (const string &mediumName, const string &libraryName, const string &substanceName, **BaseSolver** \*const solver, const int &uniqueID)  
*Constructor.*
- virtual ~**BaseTwoPhaseMedium** ()  
*Destructor.*
- void **reinitMedium** (const string &mediumName, const string &libraryName, const string &substanceName, **BaseSolver** \*const solver, const int &uniqueID)  
*Reinit medium constants.*
- int **uniqueID** () const  
*Return unique ID number.*
- string **mediumName** () const  
*Return medium name.*
- string **libraryName** () const  
*Return library name.*
- string **substanceName** () const  
*Return substance name.*
- **TwoPhaseMediumProperties** \* **properties** () const  
*Return pointer to properties.*
- **BaseSolver** \* **solver** () const  
*Return pointer to solver.*
- virtual void **setSolver** (**BaseSolver** \*const solver)  
*Set solver.*
- int **phase** () const  
*Return phase (error handling included).*

- double **beta** () const  
*Return isothermal expansion coefficient (error handling included).*
- double **cp** () const  
*Return specific heat capacity cp (error handling included).*
- double **cv** () const  
*Return specific heat capacity cv (error handling included).*
- double **d** () const  
*Return density (error handling included).*
- double **dd\_dp\_h** () const  
*Return derivative of density wrt pressure at constant specific enthalpy (error handling included).*
- double **dd\_dh\_p** () const  
*Return derivative of density wrt specific enthalpy at constant pressure (error handling included).*
- double **h** () const  
*Return specific enthalpy (error handling included).*
- double **kappa** () const  
*Return compressibility (error handling included).*
- double **p** () const  
*Return pressure (error handling included).*
- double **s** () const  
*Return specific entropy (error handling included).*
- double **T** () const  
*Return temperature (error handling included).*
- double **h\_iso** (double &p) const  
*Return the enthalpy at pressure p after an isentropic transformation from the specified medium state.*
- double **dT\_dp\_h** () const  
*Return derivative of temperature wrt pressure at constant specific enthalpy (error handling included).*
- double **dT\_dh\_p** () const  
*Return derivative of temperature wrt specific enthalpy at constant pressure (error handling included).*
- double **ps** () const  
*Return saturation pressure (error handling included).*
- double **Ts** () const

*Return saturation temperature (error handling included).*

- double **dl** () const  
*Return bubble density (error handling included).*
- double **dv** () const  
*Return dew density (error handling included).*
- double **hl** () const  
*Return bubble specific enthalpy (error handling included).*
- double **hv** () const  
*Return dew specific enthalpy (error handling included).*
- double **sl** () const  
*Return bubble specific entropy (error handling included).*
- double **sv** () const  
*Return dew specific entropy (error handling included).*
- double **d\_Ts\_dp** () const  
*Return derivative of saturation temperature wrt pressure (error handling included).*
- double **d\_dl\_dp** () const  
*Return derivative of bubble density wrt pressure (error handling included).*
- double **d\_dv\_dp** () const  
*Return derivative of dew density wrt pressure (error handling included).*
- double **d\_hl\_dp** () const  
*Return derivative of bubble specific enthalpy wrt pressure (error handling included).*
- double **d\_hv\_dp** () const  
*Return derivative of dew specific enthalpy wrt pressure (error handling included).*
- double **eta** () const  
*Return dynamic viscosity (error handling included).*
- double **lambda** () const  
*Return thermal conductivity (error handling included).*
- double **Pr** () const  
*Return Prandtl number (error handling included).*
- double **sigma** () const  
*Return surface tension (error handling included).*
- double **a** () const  
*Return velocity of sound (error handling included).*

- virtual void **setSat\_p** (double &p)  
*Set saturation properties from p.*
- virtual void **setSat\_T** (double &T)  
*Set saturation properties from T.*
- virtual void **setSat\_p\_state** ()  
*Set saturation properties.*
- virtual void **setState\_dT** (double &d, double &T, int &phase)  
*Set properties from d, T, and phase.*
- virtual void **setState\_ph** (double &p, double &h, int &phase)  
*Set properties from p, h, and phase.*
- virtual void **setState\_ps** (double &p, double &s, int &phase)  
*Set properties from p, s, and phase.*
- virtual void **setState\_pT** (double &p, double &T)  
*Set properties from p and T.*
- virtual int **getBubbleUniqueID** (int phase)  
*Get bubble unique ID.*
- virtual int **getDewUniqueID** (int phase)  
*Get dew unique ID.*
- virtual void **setDewState** (int phase)  
*Set dew state.*
- virtual void **setBubbleState** (int phase)  
*Set bubble state.*

## Protected Attributes

- **TwoPhaseMediumProperties \* \_\_properties**  
*Medium property record.*
- **BaseSolver \* \_\_solver**  
*Solver.*
- **int \_\_uniqueID**  
*Unique ID.*
- **string \_\_mediumName**  
*Medium name.*
- **string \_\_libraryName**

*Library name.*

- **string \_\_substanceName**  
*Substance name.*
- **int \_\_dewUniqueIDOnePhase**  
*Unique ID of corresponding 1-phase dew state medium object, set by setDewState.*
- **int \_\_dewUniqueIDTwoPhase**  
*Unique ID of corresponding 2-phase dew state medium object, set by setDewState.*
- **int \_\_bubbleUniqueIDOnePhase**  
*Unique ID of corresponding 1-phase bubble state medium object, set by setBubbleState.*
- **int \_\_bubbleUniqueIDTwoPhase**  
*Unique ID of corresponding 2-phase bubble state medium object, set by setBubbleState.*

### 7.2.1 Detailed Description

Base two-phase medium class.

This class defines all the variables and member functions which are needed to use external Modelica medium models extending from PartialExternalTwoPhaseMedium.

The functions defined here are not fluid-specific, thus need not be adapted to your own specific fluid property computation code.

Francesco Casella, Christoph Richter, Sep 2006 Copyright Politecnico di Milano and TU Braunschweig

### 7.2.2 Constructor & Destructor Documentation

#### 7.2.2.1 BaseTwoPhaseMedium::BaseTwoPhaseMedium (const string & *mediumName*, const string & *libraryName*, const string & *substanceName*, BaseSolver \*const *solver*, const int & *uniqueID*)

Constructor.

The constructor initializes the following variables with the specified values:

- **\_\_mediumName** = *mediumName*
- **\_\_libraryName** = *libraryName*
- **\_\_substanceName** = *substanceName*
- **\_\_solver** = *solver*
- **\_\_uniqueID** = *uniqueID*
- **\_\_dewUniqueIDOnePhase** = 0
- **\_\_dewUniqueIDTwoPhase** = 0

- `_bubbleUniqueIDOnePhase = 0`
- `_bubbleUniqueIDTwoPhase = 0`

**Parameters:**

*mediumName* Medium name  
*libraryName* Library name  
*substanceName* Substance name  
*solver* Solver  
*uniqueID* Unique ID number

### 7.2.2.2 BaseTwoPhaseMedium::~~BaseTwoPhaseMedium () [virtual]

Destructor.

The destructor for the base solver if currently not doing anything.

## 7.2.3 Member Function Documentation

### 7.2.3.1 void BaseTwoPhaseMedium::reinitMedium (const string & *mediumName*, const string & *libraryName*, const string & *substanceName*, BaseSolver \*const *solver*, const int & *uniqueID*)

Reinit medium constants.

This function reinites the medium constans to the inputs and the properties to their default values.

**Parameters:**

*mediumName* Medium name  
*libraryName* Library name  
*substanceName* Substance name  
*solver* Solver  
*uniqueID* Unique ID number

See also:

`BaseTwoPhaseMedium()` (p. 26)

### 7.2.3.2 void BaseTwoPhaseMedium::setSolver (BaseSolver \*const *solver*) [virtual]

Set solver.

This functions ets the solver.

**Parameters:**

*solver* New solver

### 7.2.3.3 void BaseTwoPhaseMedium::setSat\_p (double & *p*) [virtual]

Set saturation properties from *p*.

This function calls the **setSat\_p()** (p. 28) function of the solver.

#### Parameters:

*p* Pressure

### 7.2.3.4 void BaseTwoPhaseMedium::setSat\_T (double & *T*) [virtual]

Set saturation properties from *T*.

This function calls the **setSat\_T()** (p. 28) function of the solver.

#### Parameters:

*T* Temperature

### 7.2.3.5 void BaseTwoPhaseMedium::setSat\_p\_state () [virtual]

Set saturation properties.

This function calls the **setSat\_p\_state()** (p. 28) function of the solver and is designed to be used for function calls from within BaseProperties.

### 7.2.3.6 void BaseTwoPhaseMedium::setState\_dT (double & *d*, double & *T*, int & *phase*) [virtual]

Set properties from *d*, *T*, and *phase*.

This function calls the **setState\_dT()** (p. 28) function of the solver.

#### Parameters:

*d* Density

*T* Temperature

*phase* Phase (2 for two-phase, 1 for one-phase, 0 if not known)

### 7.2.3.7 void BaseTwoPhaseMedium::setState\_ph (double & *p*, double & *h*, int & *phase*) [virtual]

Set properties from *p*, *h*, and *phase*.

This function calls the **setState\_ph()** (p. 28) function of the solver.

#### Parameters:

*p* Pressure

*h* Specific enthalpy

*phase* Phase (2 for two-phase, 1 for one-phase, 0 if not known)



### 7.2.3.8 void BaseTwoPhaseMedium::setState\_ps (double & *p*, double & *s*, int & *phase*) [virtual]

Set properties from *p*, *s*, and *phase*.

This function calls the `setState_ps()` (p. 29) function of the solver.

#### Parameters:

- p* Pressure
- s* Specific entropy
- phase* Phase (2 for two-phase, 1 for one-phase, 0 if not known)

### 7.2.3.9 void BaseTwoPhaseMedium::setState\_pT (double & *p*, double & *T*) [virtual]

Set properties from *p* and *T*.

This function calls the `setState_pT()` (p. 29) function of the solver.

#### Parameters:

- p* Pressure
- T* Temperature

### 7.2.3.10 int BaseTwoPhaseMedium::getBubbleUniqueID (int *phase*) [virtual]

Get bubble unique ID.

Get the unique ID of the bubble point medium object corresponding to *psat*, *Tsat* and the *phase* input, possibly allocating a medium object on the medium map if needed.

#### Parameters:

- phase* Phase (2 for two-phase, 1 for one-phase, 0 if not known)

### 7.2.3.11 int BaseTwoPhaseMedium::getDewUniqueID (int *phase*) [virtual]

Get dew unique ID.

Get the unique ID of the dew point medium object corresponding to *psat*, *Tsat* and the *phase* input, possibly allocating a medium object on the medium map if needed.

#### Parameters:

- phase* Phase (2 for two-phase, 1 for one-phase, 0 if not known)

### 7.2.3.12 void BaseTwoPhaseMedium::setDewState (int *phase*) [virtual]

Set dew state.

Sets the properties of the dew medium depending on the specified *phase*.

#### Parameters:

- phase* Phase (2 for two-phase, 1 for one-phase, 0 if not known)

**7.2.3.13** void BaseTwoPhaseMedium::setBubbleState (int *phase*) [virtual]

Set bubble state.

Sets the properties of the bubble medium depending on the specified phase.

**Parameters:**

*phase* Phase (2 for two-phase, 1 for one-phase, 0 if not known)

The documentation for this class was generated from the following files:

- basetwophasemedium.h
- basetwophasemedium.cpp

## 7.3 FluidConstants Struct Reference

Fluid constants struct.

```
#include <fluidconstants.h>
```

### Public Attributes

- double **MM**  
*Molar mass.*
- double **pc**  
*Pressure at critical point.*
- double **Tc**  
*Temperature at critical point.*
- double **dc**  
*Density at critical point.*
- double **hc**  
*Specific enthalpy at critical point.*
- double **sc**  
*Specific entropy at critical point.*

### 7.3.1 Detailed Description

Fluid constants struct.

The fluid constants struct contains all the constant fluid properties that are returned by the external solver.

Francesco Casella, Christoph Richter, Nov 2006 Copyright Politecnico di Milano and TU Braunschweig

The documentation for this struct was generated from the following file:

- fluidconstants.h

## 7.4 MediumMap Class Reference

Medium map.

```
#include <mediummap.h>
```

### Static Public Member Functions

- static int **addMedium** (const string &mediumName, const string &libraryName, const string &substanceName)  
*Add a new medium object to the medium map.*
- static int **addTransientMedium** (const string &mediumName, const string &libraryName, const string &substanceName)  
*Add a new transient medium object to the medium map.*
- static void **addSolverMedium** (const string &solverKey, **BaseSolver** \*const solver)  
*Add the default medium object for the solver to the default solver media map.*
- static void **changeMedium** (const string &mediumName, const string &libraryName, const string &substanceName, const int &uniqueID)  
*Change a medium.*
- static void **deleteMedium** (const int &uniqueID)  
*Delete a medium from the map.*
- static **BaseTwoPhaseMedium** \* **medium** (const int &uniqueID)  
*Return a pointer to a medium object.*
- static **BaseTwoPhaseMedium** \* **solverMedium** (**BaseSolver** \*const solver)  
*Return a pointer to the default medium for the solver.*
- static **BaseTwoPhaseMedium** \* **solverMedium** (const string &mediumName, const string &libraryName, const string &substanceName)  
*Return a pointer to the default medium for the solver.*

### Static Protected Attributes

- static int **\_\_uniqueID**  
*Static counter for the positive unique ID number used by permanent medium objects.*
- static int **\_\_transientUniqueID**  
*Static counter for the negative unique ID number used by transient medium objects.*
- static map< int, **BaseTwoPhaseMedium** \* > **\_\_mediums**  
*Map for mediums with unique ID as identifier.*
- static map< string, **BaseTwoPhaseMedium** \* > **\_\_solverMediums**  
*Map for mediums that are used for function calls without specified unique ID such as `saturationTemperature_()` and `saturationPressure_()`.*

### 7.4.1 Detailed Description

Medium map.

This class is used to hold a static medium map. It stores instances of **TwoPhaseMedium** (p. 42) to avoid the time-consuming recomputation of fluid properties. It uses positive as well as negative unique ID number to uniquely identify the instances. The positive unique ID numbers are used for instances of the Modelica model BaseProperties. They have to be created from within Modelica when the Modelica models are initialized. The negative unique ID numbers are also called transient unique IDs and are used when the BaseProperties record is not used in Modelica. The maximum number of transient unique IDs is called MAX\_TRANSIENT\_MEDIUM and can be set in **include.h** (p. 71).

Francesco Casella, Christoph Richter, Sep 2006 Copyright Politecnico di Milano and TU Braunschweig

### 7.4.2 Member Function Documentation

#### 7.4.2.1 `int MediumMap::addMedium (const string & mediumName, const string & libraryName, const string & substanceName) [static]`

Add a new medium object to the medium map.

This function adds a new medium object to the medium map and returns its (positive) uniqueID.

##### Parameters:

*mediumName* Medium name

*libraryName* Library name

*substanceName* Substance name

#### 7.4.2.2 `int MediumMap::addTransientMedium (const string & mediumName, const string & libraryName, const string & substanceName) [static]`

Add a new transient medium object to the medium map.

This function adds a new transient medium object to the medium map and returns its (negative) uniqueID.

##### Parameters:

*mediumName* Medium name

*libraryName* Library name

*substanceName* Substance name

#### 7.4.2.3 `void MediumMap::addSolverMedium (const string & solverKey, BaseSolver *const solver) [static]`

Add the default medium object for the solver to the default solver media map.

This function adds the default medium object for the specified solver to the default solver media map. The default medium object is used for function calls without a uniqueID (i.e. all functions returning fluid constants, saturationPressure(), saturationTemperature(), etc.).

**Parameters:**

***solverKey*** Solver key  
***solver*** Pointer to solver

**7.4.2.4 void MediumMap::changeMedium (const string & *mediumName*, const string & *libraryName*, const string & *substanceName*, const int & *uniqueID*) [static]**

Change a medium.

This function replaces the existing pointer to a solver that is part of the medium object with a pointer to another solver.

**Parameters:**

***mediumName*** Medium name  
***libraryName*** Library name  
***substanceName*** Substance name  
***uniqueID*** uniqueID of the medium object for which the solver should be changed

**7.4.2.5 void MediumMap::deleteMedium (const int & *uniqueID*) [static]**

Delete a medium from the map.

This function deletes a medium object from the medium map. It can be called when the Modelica simulation is terminated.

**Parameters:**

***uniqueID*** UniqueID of the medium to be deleted

**7.4.2.6 BaseTwoPhaseMedium \* MediumMap::medium (const int & *uniqueID*) [static]**

Return a pointer to a medium object.

This function returns a pointer to a medium object in the medium map. The advantage of using this function instead of using the medium map directly is that an error reporting mechanism can be implemented that avoids crashed due to calls with an invalid uniqueID.

**Parameters:**

***uniqueID*** UniqueID of the medium object

**7.4.2.7 BaseTwoPhaseMedium \* MediumMap::solverMedium (BaseSolver \*const *solver*) [static]**

Return a pointer to the default medium for the solver.

This function returns a pointer to a medium object in the solver medium map.

**Parameters:**

***solver*** Pointer to solver

#### 7.4.2.8 BaseTwoPhaseMedium \* MediumMap::solverMedium (const string & *mediumName*, const string & *libraryName*, const string & *substanceName*) [static]

Return a pointer to the default medium for the solver.

This function returns a pointer to a medium object in the solver medium map. It is a convenience function if the pointer to the solver is not available. It creates a new solver medium if it does not exist.

##### Parameters:

*mediumName* Medium name

*libraryName* Library name

*substanceName* Substance name

The documentation for this class was generated from the following files:

- mediummap.h
- mediummap.cpp

## 7.5 SolverMap Class Reference

Solver map.

```
#include <solvermap.h>
```

### Static Public Member Functions

- static **BaseSolver** \* **getSolver** (const string &mediumName, const string &libraryName, const string &substanceName)  
*Get a specific solver.*
- static string **solverKey** (const string &libraryName, const string &substanceName)  
*Generate a unique solver key.*

### Static Protected Attributes

- static map< string, **BaseSolver** \* > **\_solvers**  
*Map for all solver instances identified by the SolverKey.*

#### 7.5.1 Detailed Description

Solver map.

This class manages the map of all solvers. A solver is a class that inherits from **BaseSolver** (p. 15) and that interfaces the external fluid property computation code. Only one instance is created for each external library.

Francesco Casella, Christoph Richter, Sep 2006 Copyright Politecnico di Milano and TU Braunschweig

#### 7.5.2 Member Function Documentation

##### 7.5.2.1 **BaseSolver** \* **SolverMap::getSolver** (const string & *mediumName*, const string & *libraryName*, const string & *substanceName*) [static]

Get a specific solver.

This function returns the solver for the specified library name, substance name and possibly medium name. It creates a new solver if the solver does not already exist. When implementing new solvers, one has to add the newly created solvers to this function. An error message is generated if the specific library is not supported by the interface library.

##### Parameters:

*mediumName* Medium name

*libraryName* Library name

*substanceName* Substance name



### 7.5.2.2 string SolverMap::solverKey (const string & *libraryName*, const string & *substanceName*) [static]

Generate a unique solver key.

This function generates a unique solver key based on the library name and substance name.

The documentation for this class was generated from the following files:

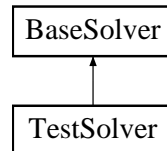
- solvermap.h
- solvermap.cpp

## 7.6 TestSolver Class Reference

Test solver class.

```
#include <testsolver.h>
```

Inheritance diagram for TestSolver::



### Public Member Functions

- virtual void **setFluidConstants** ()  
*Set fluid constants.*
- virtual void **setSat\_p** (double &p, **TwoPhaseMediumProperties** \*const properties)  
*Set saturation properties from p.*
- virtual void **setSat\_T** (double &T, **TwoPhaseMediumProperties** \*const properties)  
*Set saturation properties from T.*
- virtual void **setSat\_p\_state** (**TwoPhaseMediumProperties** \*const properties)  
*Set saturation properties from within BaseProperties model.*
- virtual void **setState\_dT** (double &d, double &T, int &phase, **TwoPhaseMediumProperties** \*const properties)  
*Set state from d, T, and phase.*
- virtual void **setState\_ph** (double &p, double &h, int &phase, **TwoPhaseMediumProperties** \*const properties)  
*Set state from p, h, and phase.*
- virtual void **setState\_ps** (double &p, double &s, int &phase, **TwoPhaseMediumProperties** \*const properties)  
*Set state from p, s, and phase.*
- virtual void **setState\_pT** (double &p, double &T, **TwoPhaseMediumProperties** \*const properties)  
*Set state from p and T.*

### 7.6.1 Detailed Description

Test solver class.

This class defines a dummy solver object, computing properties of a fluid roughly resembling warm water at low pressure, without the need of any further external code. The class is useful for

debugging purposes, to test whether the C compiler and the Modelica tools are set up correctly before tackling problems with the actual - usually way more complex - external code. It is *\*not\** meant to be used as an actual fluid model for any real application.

To keep complexity down to the absolute medium, the current version of the solver can only compute the fluid properties in the liquid phase region:  $1\text{e}5\text{ Pa} < p < 2\text{e}5\text{ Pa}$   $300\text{ K} < T < 350\text{ K}$ ; results returned with inputs outside that range (possibly corresponding to two-phase or vapour points) are not reliable. Saturation properties are computed in the range  $1\text{e}5\text{ Pa} < p_{\text{sat}} < 2\text{e}5\text{ Pa}$ ; results obtained outside that range might be unrealistic.

To instantiate this solver, it is necessary to set the library name package constant in Modelica as follows:

```
libraryName = "TestMedium";
```

Francesco Casella, Christoph Richter, Oct 2006 Copyright Politecnico di Milano and TU Braunschweig

## 7.6.2 Member Function Documentation

### 7.6.2.1 void TestSolver::setFluidConstants () [virtual]

Set fluid constants.

This function sets the fluid constants which are defined in the **FluidConstants** (p. 31) record in Modelica. It should be called when a new solver is created.

Must be re-implemented in the specific solver

Reimplemented from **BaseSolver** (p. 18).

### 7.6.2.2 void TestSolver::setSat\_p (double & p, TwoPhaseMediumProperties \*const properties) [virtual]

Set saturation properties from p.

This function sets the saturation properties for the given pressure p. The computed values are written to the two phase medium property struct.

Must be re-implemented in the specific solver

#### Parameters:

*p* Pressure

*properties* Two phase medium property record

Reimplemented from **BaseSolver** (p. 18).

### 7.6.2.3 void TestSolver::setSat\_T (double & T, TwoPhaseMediumProperties \*const properties) [virtual]

Set saturation properties from T.

This function sets the saturation properties for the given temperature T. The computed values are written to the two phase medium property struct.

Must be re-implemented in the specific solver

**Parameters:**

*T* Temperature  
*properties* Two phase medium property record

Reimplemented from **BaseSolver** (p. 18).

#### 7.6.2.4 void TestSolver::setSat\_p\_state (TwoPhaseMediumProperties \*const *properties*) [virtual]

Set saturation properties from within BaseProperties model.

This function sets the saturation properties for the given pressure p and is desined to be used from within the BaseProperties model in Modelica. The computed values are written to the two phase medium propety struct.

Must be re-implemented in the specific solver

**Parameters:**

*properties* Two phase medium property record

Reimplemented from **BaseSolver** (p. 18).

#### 7.6.2.5 void TestSolver::setState\_dT (double & *d*, double & *T*, int & *phase*, TwoPhaseMediumProperties \*const *properties*) [virtual]

Set state from d, T, and phase.

This function sets the thermodynamic state record for the given density d, the temperature T and the specified phase. The computed values are written to the two phase medium property struct.

Must be re-implemented in the specific solver

**Parameters:**

*d* Density  
*T* Temperature  
*phase* Phase (2 for two-phase, 1 for one-phase, 0 if not known)  
*properties* Two phase medium property record

Reimplemented from **BaseSolver** (p. 19).

#### 7.6.2.6 void TestSolver::setState\_ph (double & *p*, double & *h*, int & *phase*, TwoPhaseMediumProperties \*const *properties*) [virtual]

Set state from p, h, and phase.

This function sets the thermodynamic state record for the given pressure p, the specific enthalpy h and the specified phase. The computed values are written to the two phase medium property struct.

Must be re-implemented in the specific solver

**Parameters:**

*p* Pressure

*h* Specific enthalpy

*phase* Phase (2 for two-phase, 1 for one-phase, 0 if not known)

*properties* Two phase medium property record

Reimplemented from **BaseSolver** (p. 19).

**7.6.2.7** `void TestSolver::setState_ps (double & p, double & s, int & phase,  
TwoPhaseMediumProperties *const properties) [virtual]`

Set state from p, s, and phase.

This function sets the thermodynamic state record for the given pressure p, the specific entropy s and the specified phase. The computed values are written to the two phase medium property struct.

Must be re-implemented in the specific solver

**Parameters:**

*p* Pressure

*s* Specific entropy

*phase* Phase (2 for two-phase, 1 for one-phase, 0 if not known)

*properties* Two phase medium property record

Reimplemented from **BaseSolver** (p. 19).

**7.6.2.8** `void TestSolver::setState_pT (double & p, double & T,  
TwoPhaseMediumProperties *const properties) [virtual]`

Set state from p and T.

This function sets the thermodynamic state record for the given pressure p and the temperature T. The computed values are written to the two phase medium property struct.

Must be re-implemented in the specific solver

**Parameters:**

*p* Pressure

*T* Temperature

*properties* Two phase medium property record

Reimplemented from **BaseSolver** (p. 20).

The documentation for this class was generated from the following files:

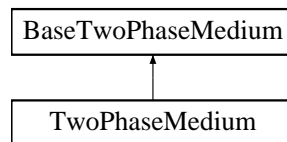
- testsolver.h
- testsolver.cpp

## 7.7 TwoPhaseMedium Class Reference

Two phase medium.

```
#include <twophasemedium.h>
```

Inheritance diagram for TwoPhaseMedium::



### Public Member Functions

- **TwoPhaseMedium** (const string &mediumName, const string &libraryName, const string &substanceName, **BaseSolver** \*const solver, const int &uniqueID)  
*Constructor.*
- **~TwoPhaseMedium** ()  
*Destructor.*

### 7.7.1 Detailed Description

Two phase medium.

This class is the default object embedding the fluid property computations at a given point of the plant.

Francesco Casella, Christoph Richter, Sep 2006 Copyright Politecnico di Milano and TU Braunschweig

### 7.7.2 Constructor & Destructor Documentation

#### 7.7.2.1 TwoPhaseMedium::TwoPhaseMedium (const string & *mediumName*, const string & *libraryName*, const string & *substanceName*, **BaseSolver** \*const *solver*, const int & *uniqueID*)

Constructor.

This constructor is passing the argument to the constructor of the base class and is creating a the property container.

**Parameters:**

*mediumName* Medium name  
*libraryName* Library name  
*substanceName* Substance name  
*solver* Solver  
*uniqueID* Unique ID number

### 7.7.2.2 TwoPhaseMedium::~~TwoPhaseMedium ()

Destructor.

The destructor is deleting the property container.

The documentation for this class was generated from the following files:

- twophasemedium.h
- twophasemedium.cpp

## 7.8 TwoPhaseMediumProperties Class Reference

Two phase medium property class.

```
#include <twophasemediumproperties.h>
```

### Public Member Functions

- **TwoPhaseMediumProperties** ()
- void **initializeFields** ()

### Public Attributes

- int **phase**  
*Phase.*
- double **beta**  
*Isobaric expansion coefficient.*
- double **cp**  
*Specific heat capacity cp.*
- double **cv**  
*Specific heat capacity cv.*
- double **d**  
*Density.*
- double **dd\_dp\_h**  
*Derivative of density wrt pressure at constant enthalpy.*
- double **dd\_dh\_p**  
*Derivative of density wrt enthalpy at constant pressure.*
- double **h**  
*Specific enthalpy.*
- double **kappa**  
*Compressibility.*
- double **p**  
*Pressure.*
- double **s**  
*Specific entropy.*
- double **T**  
*Temperature.*
- double **dT\_dp\_h**



*Derivative of temperature wrt pressure at constant enthalpy.*

- double **dT\_dh\_p**

*Derivative of temperature wrt enthalpy at constant pressure.*

- double **ps**

*Saturation pressure.*

- double **Ts**

*Saturation temperature.*

- double **dl**

*Density at bubble line (for pressure ps).*

- double **dv**

*Density at dew line (for pressure ps).*

- double **hl**

*Specific enthalpy at bubble line (for pressure ps).*

- double **hv**

*Specific enthalpy at dew line (for pressure ps).*

- double **sl**

*Specific entropy at bubble line (for pressure ps).*

- double **sv**

*Specific entropy at dew line (for pressure ps).*

- double **eta**

*Dynamic viscosity.*

- double **lambda**

*Thermal conductivity.*

- double **Pr**

*Prandtl number.*

- double **sigma**

*Surface tension.*

- double **a**

*Velocity of sound.*

- double **d\_Ts\_dp**

*Derivative of Ts wrt pressure.*

- double **d\_dl\_dp**

*Derivative of dls wrt pressure.*

- double **d\_dv\_dp**  
*Derivative of dvs wrt pressure.*
- double **d\_hl\_dp**  
*Derivative of hls wrt pressure.*
- double **d\_hv\_dp**  
*Derivative of hvs wrt pressure.*

### 7.8.1 Detailed Description

Two phase medium property class.

The two phase medium property struct defines all the properties that are computed by external Modelica medium models extending from `PartialExternalTwoPhaseMedium`. It also contains the saturation properties for the dew and the bubble line which makes it a real two phase medium property struct.

Francesco Casella, Christoph Richter, Sep 2006 Copyright Politecnico di Milano and TU Braunschweig

### 7.8.2 Constructor & Destructor Documentation

#### 7.8.2.1 `TwoPhaseMediumProperties::TwoPhaseMediumProperties ()`

The constructor is calling `initializeFields`.

### 7.8.3 Member Function Documentation

#### 7.8.3.1 `void TwoPhaseMediumProperties::initializeFields ()`

This function initializes all fields of **TwoPhaseMediumProperties** (p. 44) to the NAN value defined in **include.h** (p. 71). The phase is initialized to zero.

### 7.8.4 Member Data Documentation

#### 7.8.4.1 `int TwoPhaseMediumProperties::phase`

Phase.

This phase flag is defined according to the phase flag in `Modelica.Media`: 2 for two-phase, 1 for one-phase.

The documentation for this class was generated from the following files:

- `twophasemediumproperties.h`
- `twophasemediumproperties.cpp`

## Chapter 8

# ExternalMedia File Documentation

### 8.1 errorhandling.h File Reference

Error handling for external library.

```
#include "include.h"
```

```
#include "ModelicaUtilities.h"
```

#### Functions

- void **errorMessage** (char \*errorMessage)  
*Function to display error message.*
- void **warningMessage** (char \*warningMessage)  
*Function to display warning message.*

#### 8.1.1 Detailed Description

Error handling for external library.

Errors in the external fluid property library have to be reported to the Modelica layer. This class defines the required interface functions.

Francesco Casella, Christoph Richter, Nov 2006 Copyright Politecnico di Milano and TU Braunschweig

#### 8.1.2 Function Documentation

##### 8.1.2.1 void **errorMessage** (char \* *errorMessage*)

Function to display error message.

Calling this function will display the specified error message and will terminate the simulation.

#### Parameters:

***errorMessage*** Error message to be displayed

**8.1.2.2 void warningMessage (char \* *warningMessage*)**

Function to display warning message.

Calling this function will display the specified warning message.

**Parameters:**

*warningMessage* Warning message to be displayed

## 8.2 externalmedialib.cpp File Reference

Interface layer.

```
#include "externalmedialib.h"
#include "mediummap.h"
#include "twophasemedium.h"
```

### Functions

- void **TwoPhaseMedium\_setStateDefault\_\_** (**BaseTwoPhaseMedium** \*medium, int choice, double d, double h, double p, double s, double T, int phase)  
*Call the appropriate setState\_xx() function of the medium object.*
- int **TwoPhaseMedium\_createMedium\_\_** (const char \*mediumName, const char \*libraryName, const char \*substanceName, int oldUniqueID)  
*Create medium.*
- double **TwoPhaseMedium\_getMolarMass\_\_** (const char \*mediumName, const char \*libraryName, const char \*substanceName)  
*Get molar mass.*
- double **TwoPhaseMedium\_getCriticalTemperature\_\_** (const char \*mediumName, const char \*libraryName, const char \*substanceName)  
*Get critical temperature.*
- double **TwoPhaseMedium\_getCriticalPressure\_\_** (const char \*mediumName, const char \*libraryName, const char \*substanceName)  
*Get critical pressure.*
- double **TwoPhaseMedium\_getCriticalMolarVolume\_\_** (const char \*mediumName, const char \*libraryName, const char \*substanceName)  
*Get critical molar volume.*
- void **TwoPhaseMedium\_setState\_dT\_\_** (double d, double T, int phase, int uniqueID, int \*state\_uniqueID, int \*state\_phase, double \*state\_d, double \*state\_h, double \*state\_p, double \*state\_s, double \*state\_T, const char \*mediumName, const char \*libraryName, const char \*substanceName)  
*Compute properties from d, T, and phase.*
- void **TwoPhaseMedium\_setState\_ph\_\_** (double p, double h, int phase, int uniqueID, int \*state\_uniqueID, int \*state\_phase, double \*state\_d, double \*state\_h, double \*state\_p, double \*state\_s, double \*state\_T, const char \*mediumName, const char \*libraryName, const char \*substanceName)  
*Compute properties from p, h, and phase.*
- void **TwoPhaseMedium\_setState\_ps\_\_** (double p, double s, int phase, int uniqueID, int \*state\_uniqueID, int \*state\_phase, double \*state\_d, double \*state\_h, double \*state\_p, double \*state\_s, double \*state\_T, const char \*mediumName, const char \*libraryName, const char \*substanceName)

*Compute properties from p, s, and phase.*

- void **TwoPhaseMedium\_setState\_pT\_\_** (double p, double T, int phase, int uniqueID, int \*state\_uniqueID, int \*state\_phase, double \*state\_d, double \*state\_h, double \*state\_p, double \*state\_s, double \*state\_T, const char \*mediumName, const char \*libraryName, const char \*substanceName)

*Compute properties from p, T, and phase.*

- void **TwoPhaseMedium\_setSat\_p\_\_** (double p, int uniqueID, double \*sat\_psat, double \*sat\_Tsat, int \*sat\_uniqueID, const char \*mediumName, const char \*libraryName, const char \*substanceName)

*Compute saturation properties from p.*

- void **TwoPhaseMedium\_setSat\_T\_\_** (double T, int uniqueID, double \*sat\_psat, double \*sat\_Tsat, int \*sat\_uniqueID, const char \*mediumName, const char \*libraryName, const char \*substanceName)

*Compute saturation properties from T.*

- void **TwoPhaseMedium\_setSat\_p\_state\_\_** (int uniqueID, double \*sat\_psat, double \*sat\_Tsat, int \*sat\_uniqueID)

*Compute saturation properties from within BaseProperties.*

- void **TwoPhaseMedium\_setDewState\_\_** (int uniqueID, int phase, int \*state\_uniqueID, int \*state\_phase, const char \*mediumName, const char \*libraryName, const char \*substanceName)

*Compute dew state.*

- void **TwoPhaseMedium\_setBubbleState\_\_** (int uniqueID, int phase, int \*state\_uniqueID, int \*state\_phase, const char \*mediumName, const char \*libraryName, const char \*substanceName)

*Compute bubble state.*

- double **TwoPhaseMedium\_density\_\_** (int uniqueID, int choice, double d, double h, double p, double s, double T, int phase, const char \*mediumName, const char \*libraryName, const char \*substanceName)

*Return density of specified medium.*

- double **TwoPhaseMedium\_density\_derp\_h\_\_** (int uniqueID, int choice, double d, double h, double p, double s, double T, int phase, const char \*mediumName, const char \*libraryName, const char \*substanceName)

*Return derivative of density wrt pressure at constant specific enthalpy of specified medium.*

- double **TwoPhaseMedium\_density\_derh\_p\_\_** (int uniqueID, int choice, double d, double h, double p, double s, double T, int phase, const char \*mediumName, const char \*libraryName, const char \*substanceName)

*Return derivative of density wrt specific enthalpy at constant pressure of specified medium.*

- double **TwoPhaseMedium\_density\_ph\_der\_\_** (int uniqueID, double p\_der, double h\_der, double p, double h, int phase, const char \*mediumName, const char \*libraryName, const char \*substanceName)

*Return derivative of density wrt pressure and specific enthalpy of specified medium.*

- double **TwoPhaseMedium\_pressure\_\_** (int uniqueID, int choice, double d, double h, double p, double s, double T, int phase, const char \*mediumName, const char \*libraryName, const char \*substanceName)  
*Return pressure of specified medium.*
- double **TwoPhaseMedium\_specificEnthalpy\_\_** (int uniqueID, int choice, double d, double h, double p, double s, double T, int phase, const char \*mediumName, const char \*libraryName, const char \*substanceName)  
*Return specific enthalpy of specified medium.*
- double **TwoPhaseMedium\_specificEntropy\_\_** (int uniqueID, int choice, double d, double h, double p, double s, double T, int phase, const char \*mediumName, const char \*libraryName, const char \*substanceName)  
*Return specific entropy of specified medium.*
- double **TwoPhaseMedium\_temperature\_\_** (int uniqueID, int choice, double d, double h, double p, double s, double T, int phase, const char \*mediumName, const char \*libraryName, const char \*substanceName)  
*Return temperature of specified medium.*
- double **TwoPhaseMedium\_temperature\_ph\_der\_\_** (int uniqueID, double p\_der, double h\_der, double p, double h, int phase, const char \*mediumName, const char \*libraryName, const char \*substanceName)  
*Return derivative of temperature wrt pressure and specific enthalpy of specified medium.*
- double **TwoPhaseMedium\_isentropicEnthalpy\_\_** (double p\_iso, int uniqueID, int choice, double d, double h, double p, double s, double T, int phase, const char \*mediumName, const char \*libraryName, const char \*substanceName)  
*Return the enthalpy at pressure p after an isentropic transformation from the specified medium state.*
- double **TwoPhaseMedium\_saturationTemperature\_derp\_sat\_\_** (double psat, double Tsat, int uniqueID, const char \*mediumName, const char \*libraryName, const char \*substanceName)  
*Return derivative of saturation temperature of specified medium from saturation properties.*
- double **TwoPhaseMedium\_bubbleDensity\_\_** (double psat, double Tsat, int uniqueID, const char \*mediumName, const char \*libraryName, const char \*substanceName)  
*Return bubble density of specified medium from saturation properties.*
- double **TwoPhaseMedium\_dewDensity\_\_** (double psat, double Tsat, int uniqueID, const char \*mediumName, const char \*libraryName, const char \*substanceName)  
*Return dew density of specified medium from saturation properties.*
- double **TwoPhaseMedium\_bubbleEnthalpy\_\_** (double psat, double Tsat, int uniqueID, const char \*mediumName, const char \*libraryName, const char \*substanceName)  
*Return bubble specific enthalpy of specified medium from saturation properties.*
- double **TwoPhaseMedium\_dewEnthalpy\_\_** (double psat, double Tsat, int uniqueID, const char \*mediumName, const char \*libraryName, const char \*substanceName)

*Return dew specific enthalpy of specified medium from saturation properties.*

- double **TwoPhaseMedium\_bubbleEntropy\_** (double psat, double Tsat, int uniqueID, const char \*mediumName, const char \*libraryName, const char \*substanceName)

*Return bubble specific entropy of specified medium from saturation properties.*

- double **TwoPhaseMedium\_dewEntropy\_** (double psat, double Tsat, int uniqueID, const char \*mediumName, const char \*libraryName, const char \*substanceName)

*Return dew specific entropy of specified medium from saturation properties.*

- double **TwoPhaseMedium\_dBubbleDensity\_dPressure\_** (double psat, double Tsat, int uniqueID, const char \*mediumName, const char \*libraryName, const char \*substanceName)

*Return derivative of bubble density wrt pressure of specified medium from saturation properties.*

- double **TwoPhaseMedium\_dDewDensity\_dPressure\_** (double psat, double Tsat, int uniqueID, const char \*mediumName, const char \*libraryName, const char \*substanceName)

*Return derivative of dew density wrt pressure of specified medium from saturation properties.*

- double **TwoPhaseMedium\_dBubbleEnthalpy\_dPressure\_** (double psat, double Tsat, int uniqueID, const char \*mediumName, const char \*libraryName, const char \*substanceName)

*Return derivative of bubble specific enthalpy wrt pressure of specified medium from saturation properties.*

- double **TwoPhaseMedium\_dDewEnthalpy\_dPressure\_** (double psat, double Tsat, int uniqueID, const char \*mediumName, const char \*libraryName, const char \*substanceName)

*Return derivative of dew specific enthalpy wrt pressure of specified medium from saturation properties.*

- double **TwoPhaseMedium\_isobaricExpansionCoefficient\_** (int uniqueID, int choice, double d, double h, double p, double s, double T, int phase, const char \*mediumName, const char \*libraryName, const char \*substanceName)

*Return isobaric expansion coefficient of specified medium.*

- double **TwoPhaseMedium\_isothermalCompressibility\_** (int uniqueID, int choice, double d, double h, double p, double s, double T, int phase, const char \*mediumName, const char \*libraryName, const char \*substanceName)

*Return isothermal compressibility of specified medium.*

- double **TwoPhaseMedium\_specificHeatCapacityCp\_** (int uniqueID, int choice, double d, double h, double p, double s, double T, int phase, const char \*mediumName, const char \*libraryName, const char \*substanceName)

*Return specific heat capacity cp of specified medium.*

- double **TwoPhaseMedium\_specificHeatCapacityCv\_** (int uniqueID, int choice, double d, double h, double p, double s, double T, int phase, const char \*mediumName, const char \*libraryName, const char \*substanceName)

*Return specific heat capacity cv of specified medium.*



- double **TwoPhaseMedium\_dynamicViscosity\_\_** (int uniqueID, int choice, double d, double h, double p, double s, double T, int phase, const char \*mediumName, const char \*libraryName, const char \*substanceName)  
*Return dynamic viscosity of specified medium.*
- double **TwoPhaseMedium\_thermalConductivity\_\_** (int uniqueID, int choice, double d, double h, double p, double s, double T, int phase, const char \*mediumName, const char \*libraryName, const char \*substanceName)  
*Return thermal conductivity of specified medium.*
- double **TwoPhaseMedium\_prandtlNumber\_\_** (int uniqueID, int choice, double d, double h, double p, double s, double T, int phase, const char \*mediumName, const char \*libraryName, const char \*substanceName)  
*Return Prandtl number of specified medium.*
- double **TwoPhaseMedium\_surfaceTension\_\_** (double psat, double Tsat, int uniqueID, const char \*mediumName, const char \*libraryName, const char \*substanceName)  
*Return surface tension of specified medium.*
- double **TwoPhaseMedium\_velocityOfSound\_\_** (int uniqueID, int choice, double d, double h, double p, double s, double T, int phase, const char \*mediumName, const char \*libraryName, const char \*substanceName)  
*Return velocity of sound of specified medium.*
- double **TwoPhaseMedium\_dDensity\_dPressure\_h\_\_** (int uniqueID, int choice, double d, double h, double p, double s, double T, int phase, const char \*mediumName, const char \*libraryName, const char \*substanceName)  
*Return derivative of density wrt pressure at constant specific enthalpy of specified medium.*
- double **TwoPhaseMedium\_dDensity\_dEnthalpy\_p\_\_** (int uniqueID, int choice, double d, double h, double p, double s, double T, int phase, const char \*mediumName, const char \*libraryName, const char \*substanceName)  
*Return derivative of density wrt specific enthalpy at constant pressure of specified medium.*
- double **TwoPhaseMedium\_saturationPressure\_\_** (double T, const char \*mediumName, const char \*libraryName, const char \*substanceName)  
*Compute saturation pressure for specified medium and temperature.*
- double **TwoPhaseMedium\_saturationTemperature\_\_** (double p, const char \*mediumName, const char \*libraryName, const char \*substanceName)  
*Compute saturation temperature for specified medium and pressure.*
- double **TwoPhaseMedium\_saturationTemperature\_derp\_\_** (double p, const char \*mediumName, const char \*libraryName, const char \*substanceName)  
*Compute derivative of saturation temperature for specified medium and pressure.*

### 8.2.1 Detailed Description

Interface layer.

C/C++ layer for external medium models extending from PartialExternalTwoPhaseMedium.

Francesco Casella, Christoph Richter, Sep 2006

## 8.2.2 Function Documentation

### 8.2.2.1 `int TwoPhaseMedium_createMedium_ (const char * mediumName, const char * libraryName, const char * substanceName, int oldUniqueID)`

Create medium.

This function creates a new medium with the specified medium name, library name, and substance name. The old unique ID is required to check whether a medium has already been created.

#### Parameters:

*mediumName* Medium name  
*libraryName* Library name  
*substanceName* Substance name  
*oldUniqueID* Old unique ID number

### 8.2.2.2 `double TwoPhaseMedium_getCriticalMolarVolume_ (const char * mediumName, const char * libraryName, const char * substanceName)`

Get critical molar volume.

This function returns the critical molar volume of the specified medium.

#### Parameters:

*mediumName* Medium name  
*libraryName* Library name  
*substanceName* Substance name

### 8.2.2.3 `double TwoPhaseMedium_getCriticalPressure_ (const char * mediumName, const char * libraryName, const char * substanceName)`

Get critical pressure.

This function returns the critical pressure of the specified medium.

#### Parameters:

*mediumName* Medium name  
*libraryName* Library name  
*substanceName* Substance name

### 8.2.2.4 `double TwoPhaseMedium_getCriticalTemperature_ (const char * mediumName, const char * libraryName, const char * substanceName)`

Get critical temperature.

This function returns the critical temperature of the specified medium.

**Parameters:**

*mediumName* Medium name  
*libraryName* Library name  
*substanceName* Substance name

**8.2.2.5** `double TwoPhaseMedium_getMolarMass_ (const char * mediumName,  
const char * libraryName, const char * substanceName)`

Get molar mass.

This function returns the molar mass of the specified medium.

**Parameters:**

*mediumName* Medium name  
*libraryName* Library name  
*substanceName* Substance name

**8.2.2.6** `void TwoPhaseMedium_setBubbleState_ (int uniqueID, int phase, int *  
state_uniqueID, int * state_phase, const char * mediumName, const char *  
libraryName, const char * substanceName)`

Compute bubble state.

This function computes the bubble state for the specified medium.

**Parameters:**

*uniqueID* Unique ID number  
*phase* Phase (2 for two-phase, 1 for one-phase, 0 if not known)  
*state\_uniqueID* Pointer to return unique ID number for state record  
*state\_phase* Pointer to return phase for state record  
*mediumName* Medium name  
*libraryName* Library name  
*substanceName* Substance name

**8.2.2.7** `void TwoPhaseMedium_setDewState_ (int uniqueID, int phase, int *  
state_uniqueID, int * state_phase, const char * mediumName, const char *  
libraryName, const char * substanceName)`

Compute dew state.

This function computes the dew state for the specified medium.

**Parameters:**

*uniqueID* Unique ID number  
*phase* Phase (2 for two-phase, 1 for one-phase, 0 if not known)  
*state\_uniqueID* Pointer to return unique ID number for state record

*state\_phase* Pointer to return phase for state record

*mediumName* Medium name

*libraryName* Library name

*substanceName* Substance name

**8.2.2.8** void TwoPhaseMedium\_setSat\_p\_ (double *p*, int *uniqueID*, double \* *sat\_psat*, double \* *sat\_Tsat*, int \* *sat\_uniqueID*, const char \* *mediumName*, const char \* *libraryName*, const char \* *substanceName*)

Compute saturation properties from *p*.

This function computes the saturation properties for the specified inputs. If the function is called with *uniqueID* == 0 a new transient unique ID is assigned and the function is called again with this new transient unique ID number.

**Parameters:**

*p* Pressure

*uniqueID* Unique ID number

*sat\_psat* Pointer to return pressure for saturation record

*sat\_Tsat* Pointer to return temperature for saturation record

*sat\_uniqueID* Pointer to return unique ID number for saturation record

*mediumName* Medium name

*libraryName* Library name

*substanceName* Substance name

**8.2.2.9** void TwoPhaseMedium\_setSat\_p\_state\_ (int *uniqueID*, double \* *sat\_psat*, double \* *sat\_Tsat*, int \* *sat\_uniqueID*)

Compute saturation properties from within BaseProperties.

This function computes the saturation properties and is designed to be called from within the BaseProperties model. The saturation properties are set according to the medium pressure

**Parameters:**

*uniqueID* Unique ID number

*sat\_psat* Pointer to return pressure for saturation record

*sat\_Tsat* Pointer to return temperature for saturation record

*sat\_uniqueID* Pointer to return unique ID number for saturation record

**8.2.2.10** void TwoPhaseMedium\_setSat\_T\_ (double *T*, int *uniqueID*, double \* *sat\_psat*, double \* *sat\_Tsat*, int \* *sat\_uniqueID*, const char \* *mediumName*, const char \* *libraryName*, const char \* *substanceName*)

Compute saturation properties from *T*.

This function computes the saturation properties for the specified inputs. If the function is called with *uniqueID* == 0 a new transient unique ID is assigned and the function is called again with this new transient unique ID number.

**Parameters:**

*T* Temperature  
*uniqueID* Unique ID number  
*sat\_psat* Pointer to return pressure for saturation record  
*sat\_Tsat* Pointer to return temperature for saturation record  
*sat\_uniqueID* Pointer to return unique ID number for saturation record  
*mediumName* Medium name  
*libraryName* Library name  
*substanceName* Substance name

**8.2.2.11** void TwoPhaseMedium\_setState\_dT\_ (double *d*, double *T*, int *phase*, int *uniqueID*, int \* *state\_uniqueID*, int \* *state\_phase*, double \* *state\_d*, double \* *state\_h*, double \* *state\_p*, double \* *state\_s*, double \* *state\_T*, const char \* *mediumName*, const char \* *libraryName*, const char \* *substanceName*)

Compute properties from *d*, *T*, and *phase*.

This function computes the properties for the specified inputs. If the function is called with *uniqueID* == 0 a new transient unique ID is assigned and the function is called again with this new transient unique ID number.

**Parameters:**

*d* Density  
*T* Temperature  
*phase* Phase (2 for two-phase, 1 for one-phase, 0 if not known)  
*uniqueID* Unique ID number  
*state\_uniqueID* Pointer to return unique ID number for state record  
*state\_phase* Pointer to return phase for state record  
*state\_d* Pointer to return density for state record  
*state\_h* Pointer to return specific enthalpy for state record  
*state\_p* Pointer to return pressure for state record  
*state\_s* Pointer to return specific entropy for state record  
*state\_T* Pointer to return temperature for state record  
*mediumName* Medium name  
*libraryName* Library name  
*substanceName* Substance name

**8.2.2.12** void TwoPhaseMedium\_setState\_ph\_ (double *p*, double *h*, int *phase*, int *uniqueID*, int \* *state\_uniqueID*, int \* *state\_phase*, double \* *state\_d*, double \* *state\_h*, double \* *state\_p*, double \* *state\_s*, double \* *state\_T*, const char \* *mediumName*, const char \* *libraryName*, const char \* *substanceName*)

Compute properties from *p*, *h*, and *phase*.

This function computes the properties for the specified inputs. If the function is called with `uniqueID == 0` a new transient unique ID is assigned and the function is called again with this new transient unique ID number.

**Parameters:**

*p* Pressure  
*h* Specific enthalpy  
*phase* Phase (2 for two-phase, 1 for one-phase, 0 if not known)  
*uniqueID* Unique ID number  
*state\_uniqueID* Pointer to return unique ID number for state record  
*state\_phase* Pointer to return phase for state record  
*state\_d* Pointer to return density for state record  
*state\_h* Pointer to return specific enthalpy for state record  
*state\_p* Pointer to return pressure for state record  
*state\_s* Pointer to return specific entropy for state record  
*state\_T* Pointer to return temperature for state record  
*mediumName* Medium name  
*libraryName* Library name  
*substanceName* Substance name

**8.2.2.13** `void TwoPhaseMedium_setState_ps_ (double p, double s, int phase, int uniqueID, int * state_uniqueID, int * state_phase, double * state_d, double * state_h, double * state_p, double * state_s, double * state_T, const char * mediumName, const char * libraryName, const char * substanceName)`

Compute properties from *p*, *s*, and *phase*.

This function computes the properties for the specified inputs. If the function is called with `uniqueID == 0` a new transient unique ID is assigned and the function is called again with this new transient unique ID number.

**Parameters:**

*p* Pressure  
*s* Specific entropy  
*phase* Phase (2 for two-phase, 1 for one-phase, 0 if not known)  
*uniqueID* Unique ID number  
*state\_uniqueID* Pointer to return unique ID number for state record  
*state\_phase* Pointer to return phase for state record  
*state\_d* Pointer to return density for state record  
*state\_h* Pointer to return specific enthalpy for state record  
*state\_p* Pointer to return pressure for state record  
*state\_s* Pointer to return specific entropy for state record  
*state\_T* Pointer to return temperature for state record  
*mediumName* Medium name  
*libraryName* Library name  
*substanceName* Substance name

**8.2.2.14** `void TwoPhaseMedium_setState_pT_ (double p, double T, int phase, int uniqueID, int * state_uniqueID, int * state_phase, double * state_d, double * state_h, double * state_p, double * state_s, double * state_T, const char * mediumName, const char * libraryName, const char * substanceName)`

Compute properties from *p*, *T*, and *phase*.

This function computes the properties for the specified inputs. If the function is called with `uniqueID == 0` a new transient unique ID is assigned and the function is called again with this new transient unique ID number.

Attention: The phase input is ignored for this function!

**Parameters:**

*p* Pressure  
*T* Temperature  
*phase* Phase (2 for two-phase, 1 for one-phase, 0 if not known)  
*uniqueID* Unique ID number  
*state\_uniqueID* Pointer to return unique ID number for state record  
*state\_phase* Pointer to return phase for state record  
*state\_d* Pointer to return density for state record  
*state\_h* Pointer to return specific enthalpy for state record  
*state\_p* Pointer to return pressure for state record  
*state\_s* Pointer to return specific entropy for state record  
*state\_T* Pointer to return temperature for state record  
*mediumName* Medium name  
*libraryName* Library name  
*substanceName* Substance name

**8.2.2.15** `void TwoPhaseMedium_setStateDefault_ (BaseTwoPhaseMedium * medium, int choice, double d, double h, double p, double s, double T, int phase)`

Call the appropriate `setState_xx()` function of the medium object.

This function calls the `setState_xx()` function of the medium object corresponding to the choice input.

**Parameters:**

*medium* Medium object (class `BaseTwoPhaseMedium` (p. 22))  
*choice* of inputs (*dT*, *ph*, *ps*, *pT*)  
*d* Density  
*h* Specific Enthalpy  
*p* Pressure  
*s* Specific entropy  
*T* Temperature  
*phase* Phase (2 for two-phase, 1 for one-phase, 0 if not known)

## 8.3 externalmedialib.h File Reference

Header file to be included in Dymola.

### Functions

- EXPORT int **TwoPhaseMedium\_createMedium\_\_** (const char \*mediumName, const char \*libraryName, const char \*substanceName, int oldUniqueID)  
*Create medium.*
- EXPORT double **TwoPhaseMedium\_getMolarMass\_\_** (const char \*mediumName, const char \*libraryName, const char \*substanceName)  
*Get molar mass.*
- EXPORT double **TwoPhaseMedium\_getCriticalTemperature\_\_** (const char \*mediumName, const char \*libraryName, const char \*substanceName)  
*Get critical temperature.*
- EXPORT double **TwoPhaseMedium\_getCriticalPressure\_\_** (const char \*mediumName, const char \*libraryName, const char \*substanceName)  
*Get critical pressure.*
- EXPORT double **TwoPhaseMedium\_getCriticalMolarVolume\_\_** (const char \*mediumName, const char \*libraryName, const char \*substanceName)  
*Get critical molar volume.*
- EXPORT void **TwoPhaseMedium\_setSat\_p\_\_** (double p, int uniqueID, double \*sat\_\_psat, double \*sat\_\_Tsat, int \*sat\_\_uniqueID, const char \*mediumName, const char \*libraryName, const char \*substanceName)  
*Compute saturation properties from p.*
- EXPORT void **TwoPhaseMedium\_setSat\_T\_\_** (double T, int uniqueID, double \*sat\_\_psat, double \*sat\_\_Tsat, int \*sat\_\_uniqueID, const char \*mediumName, const char \*libraryName, const char \*substanceName)  
*Compute saturation properties from T.*
- EXPORT void **TwoPhaseMedium\_setSat\_p\_state\_\_** (int uniqueID, double \*sat\_\_psat, double \*sat\_\_Tsat, int \*sat\_\_uniqueID)  
*Compute saturation properties from within BaseProperties.*
- EXPORT void **TwoPhaseMedium\_setDewState\_\_** (int uniqueID, int phase, int \*state\_\_uniqueID, int \*state\_\_phase, const char \*mediumName, const char \*libraryName, const char \*substanceName)  
*Compute dew state.*
- EXPORT void **TwoPhaseMedium\_setBubbleState\_\_** (int uniqueID, int phase, int \*state\_\_uniqueID, int \*state\_\_phase, const char \*mediumName, const char \*libraryName, const char \*substanceName)  
*Compute bubble state.*



- EXPORT double **TwoPhaseMedium\_saturationPressure\_\_** (double T, const char \*mediumName, const char \*libraryName, const char \*substanceName)  
*Compute saturation pressure for specified medium and temperature.*
- EXPORT double **TwoPhaseMedium\_saturationTemperature\_\_** (double p, const char \*mediumName, const char \*libraryName, const char \*substanceName)  
*Compute saturation temperature for specified medium and pressure.*
- EXPORT double **TwoPhaseMedium\_saturationTemperature\_derp\_sat\_\_** (double psat, double Tsat, int uniqueID, const char \*mediumName, const char \*libraryName, const char \*substanceName)  
*Return derivative of saturation temperature of specified medium from saturation properties.*
- EXPORT double **TwoPhaseMedium\_bubbleDensity\_\_** (double psat, double Tsat, int uniqueID, const char \*mediumName, const char \*libraryName, const char \*substanceName)  
*Return bubble density of specified medium from saturation properties.*
- EXPORT double **TwoPhaseMedium\_dewDensity\_\_** (double psat, double Tsat, int uniqueID, const char \*mediumName, const char \*libraryName, const char \*substanceName)  
*Return dew density of specified medium from saturation properties.*
- EXPORT double **TwoPhaseMedium\_bubbleEnthalpy\_\_** (double psat, double Tsat, int uniqueID, const char \*mediumName, const char \*libraryName, const char \*substanceName)  
*Return bubble specific enthalpy of specified medium from saturation properties.*
- EXPORT double **TwoPhaseMedium\_dewEnthalpy\_\_** (double psat, double Tsat, int uniqueID, const char \*mediumName, const char \*libraryName, const char \*substanceName)  
*Return dew specific enthalpy of specified medium from saturation properties.*
- EXPORT double **TwoPhaseMedium\_bubbleEntropy\_\_** (double psat, double Tsat, int uniqueID, const char \*mediumName, const char \*libraryName, const char \*substanceName)  
*Return bubble specific entropy of specified medium from saturation properties.*
- EXPORT double **TwoPhaseMedium\_dewEntropy\_\_** (double psat, double Tsat, int uniqueID, const char \*mediumName, const char \*libraryName, const char \*substanceName)  
*Return dew specific entropy of specified medium from saturation properties.*
- EXPORT double **TwoPhaseMedium\_dBubbleDensity\_dPressure\_\_** (double psat, double Tsat, int uniqueID, const char \*mediumName, const char \*libraryName, const char \*substanceName)  
*Return derivative of bubble density wrt pressure of specified medium from saturation properties.*
- EXPORT double **TwoPhaseMedium\_dDewDensity\_dPressure\_\_** (double psat, double Tsat, int uniqueID, const char \*mediumName, const char \*libraryName, const char \*substanceName)  
*Return derivative of dew density wrt pressure of specified medium from saturation properties.*
- EXPORT double **TwoPhaseMedium\_dBubbleEnthalpy\_dPressure\_\_** (double psat, double Tsat, int uniqueID, const char \*mediumName, const char \*libraryName, const char \*substanceName)

*Return derivative of bubble specific enthalpy wrt pressure of specified medium from saturation properties.*

- EXPORT double **TwoPhaseMedium\_dDewEnthalpy\_dPressure\_** (double psat, double Tsat, int uniqueID, const char \*mediumName, const char \*libraryName, const char \*substanceName)

*Return derivative of dew specific enthalpy wrt pressure of specified medium from saturation properties.*

- EXPORT void **TwoPhaseMedium\_setState\_dT\_** (double d, double T, int phase, int uniqueID, int \*state\_uniqueID, int \*state\_phase, double \*state\_d, double \*state\_h, double \*state\_p, double \*state\_s, double \*state\_T, const char \*mediumName, const char \*libraryName, const char \*substanceName)

*Compute properties from d, T, and phase.*

- EXPORT void **TwoPhaseMedium\_setState\_ph\_** (double p, double h, int phase, int uniqueID, int \*state\_uniqueID, int \*state\_phase, double \*state\_d, double \*state\_h, double \*state\_p, double \*state\_s, double \*state\_T, const char \*mediumName, const char \*libraryName, const char \*substanceName)

*Compute properties from p, h, and phase.*

- EXPORT void **TwoPhaseMedium\_setState\_ps\_** (double p, double s, int phase, int uniqueID, int \*state\_uniqueID, int \*state\_phase, double \*state\_d, double \*state\_h, double \*state\_p, double \*state\_s, double \*state\_T, const char \*mediumName, const char \*libraryName, const char \*substanceName)

*Compute properties from p, s, and phase.*

- EXPORT void **TwoPhaseMedium\_setState\_pT\_** (double p, double T, int phase, int uniqueID, int \*state\_uniqueID, int \*state\_phase, double \*state\_d, double \*state\_h, double \*state\_p, double \*state\_s, double \*state\_T, const char \*mediumName, const char \*libraryName, const char \*substanceName)

*Compute properties from p, T, and phase.*

- EXPORT double **TwoPhaseMedium\_density\_** (int uniqueID, int choice, double d, double h, double p, double s, double T, int phase, const char \*mediumName, const char \*libraryName, const char \*substanceName)

*Return density of specified medium.*

- EXPORT double **TwoPhaseMedium\_density\_ph\_der\_** (int uniqueID, double p\_der, double h\_der, double p, double h, int phase, const char \*mediumName, const char \*libraryName, const char \*substanceName)

*Return derivative of density wrt pressure and specific enthalpy of specified medium.*

- EXPORT double **TwoPhaseMedium\_density\_derp\_h\_** (int uniqueID, int choice, double d, double h, double p, double s, double T, int phase, const char \*mediumName, const char \*libraryName, const char \*substanceName)

*Return derivative of density wrt pressure at constant specific enthalpy of specified medium.*

- EXPORT double **TwoPhaseMedium\_density\_derh\_p\_** (int uniqueID, int choice, double d, double h, double p, double s, double T, int phase, const char \*mediumName, const char \*libraryName, const char \*substanceName)

*Return derivative of density wrt specific enthalpy at constant pressure of specified medium.*

- EXPORT double **TwoPhaseMedium\_\_pressure\_\_** (int uniqueID, int choice, double d, double h, double p, double s, double T, int phase, const char \*mediumName, const char \*libraryName, const char \*substanceName)  
*Return pressure of specified medium.*
- EXPORT double **TwoPhaseMedium\_\_specificEnthalpy\_\_** (int uniqueID, int choice, double d, double h, double p, double s, double T, int phase, const char \*mediumName, const char \*libraryName, const char \*substanceName)  
*Return specific enthalpy of specified medium.*
- EXPORT double **TwoPhaseMedium\_\_specificEntropy\_\_** (int uniqueID, int choice, double d, double h, double p, double s, double T, int phase, const char \*mediumName, const char \*libraryName, const char \*substanceName)  
*Return specific entropy of specified medium.*
- EXPORT double **TwoPhaseMedium\_\_temperature\_\_** (int uniqueID, int choice, double d, double h, double p, double s, double T, int phase, const char \*mediumName, const char \*libraryName, const char \*substanceName)  
*Return temperature of specified medium.*
- EXPORT double **TwoPhaseMedium\_\_isobaricExpansionCoefficient\_\_** (int uniqueID, int choice, double d, double h, double p, double s, double T, int phase, const char \*mediumName, const char \*libraryName, const char \*substanceName)  
*Return isobaric expansion coefficient of specified medium.*
- EXPORT double **TwoPhaseMedium\_\_isothermalCompressibility\_\_** (int uniqueID, int choice, double d, double h, double p, double s, double T, int phase, const char \*mediumName, const char \*libraryName, const char \*substanceName)  
*Return isothermal compressibility of specified medium.*
- EXPORT double **TwoPhaseMedium\_\_specificHeatCapacityCp\_\_** (int uniqueID, int choice, double d, double h, double p, double s, double T, int phase, const char \*mediumName, const char \*libraryName, const char \*substanceName)  
*Return specific heat capacity cp of specified medium.*
- EXPORT double **TwoPhaseMedium\_\_specificHeatCapacityCv\_\_** (int uniqueID, int choice, double d, double h, double p, double s, double T, int phase, const char \*mediumName, const char \*libraryName, const char \*substanceName)  
*Return specific heat capacity cv of specified medium.*
- EXPORT double **TwoPhaseMedium\_\_isentropicEnthalpy\_\_** (double p\_iso, int uniqueID, int choice, double d, double h, double p, double s, double T, int phase, const char \*mediumName, const char \*libraryName, const char \*substanceName)  
*Return the enthalpy at pressure p after an isentropic transformation from the specified medium state.*
- EXPORT double **TwoPhaseMedium\_\_dynamicViscosity\_\_** (int uniqueID, int choice, double d, double h, double p, double s, double T, int phase, const char \*mediumName, const char \*libraryName, const char \*substanceName)  
*Return dynamic viscosity of specified medium.*

- EXPORT double **TwoPhaseMedium\_thermalConductivity\_** (int uniqueID, int choice, double d, double h, double p, double s, double T, int phase, const char \*mediumName, const char \*libraryName, const char \*substanceName)  
*Return thermal conductivity of specified medium.*
- EXPORT double **TwoPhaseMedium\_prandtlNumber\_** (int uniqueID, int choice, double d, double h, double p, double s, double T, int phase, const char \*mediumName, const char \*libraryName, const char \*substanceName)  
*Return Prandtl number of specified medium.*
- EXPORT double **TwoPhaseMedium\_velocityOfSound\_** (int uniqueID, int choice, double d, double h, double p, double s, double T, int phase, const char \*mediumName, const char \*libraryName, const char \*substanceName)  
*Return velocity of sound of specified medium.*
- EXPORT double **TwoPhaseMedium\_dDensity\_dPressure\_h\_** (int uniqueID, int choice, double d, double h, double p, double s, double T, int phase, const char \*mediumName, const char \*libraryName, const char \*substanceName)  
*Return derivative of density wrt pressure at constant specific enthalpy of specified medium.*
- EXPORT double **TwoPhaseMedium\_dDensity\_dEnthalpy\_p\_** (int uniqueID, int choice, double d, double h, double p, double s, double T, int phase, const char \*mediumName, const char \*libraryName, const char \*substanceName)  
*Return derivative of density wrt specific enthalpy at constant pressure of specified medium.*
- EXPORT double **TwoPhaseMedium\_temperature\_ph\_der\_** (int uniqueID, double p\_der, double h\_der, double p, double h, int phase, const char \*mediumName, const char \*libraryName, const char \*substanceName)  
*Return derivative of temperature wrt pressure and specific enthalpy of specified medium.*

### 8.3.1 Detailed Description

Header file to be included in Dymola.

This is the header file to be included in the Dymola/Source directory. It provided function prototypes for all the external functions needed by medium models extending from PartialExternalTwoPhaseMedium.

Please be aware that other Modelica tools might require a slightly different header files depending on the compiler.

Francesco Casella, Christoph Richter, Sep 2006 Copyright Politecnico di Milano and TU Braunschweig

### 8.3.2 Function Documentation

- #### 8.3.2.1 EXPORT int TwoPhaseMedium\_createMedium\_ (const char \* mediumName, const char \* libraryName, const char \* substanceName, int oldUniqueID)

Create medium.

This function creates a new medium with the specified medium name, library name, and substance name. The old unique ID is required to check whether a medium has already been created.

**Parameters:**

*mediumName* Medium name  
*libraryName* Library name  
*substanceName* Substance name  
*oldUniqueID* Old unique ID number

**8.3.2.2 EXPORT double TwoPhaseMedium\_getCriticalMolarVolume\_ (const char \* *mediumName*, const char \* *libraryName*, const char \* *substanceName*)**

Get critical molar volume.

This function returns the critical molar volume of the specified medium.

**Parameters:**

*mediumName* Medium name  
*libraryName* Library name  
*substanceName* Substance name

**8.3.2.3 EXPORT double TwoPhaseMedium\_getCriticalPressure\_ (const char \* *mediumName*, const char \* *libraryName*, const char \* *substanceName*)**

Get critical pressure.

This function returns the critical pressure of the specified medium.

**Parameters:**

*mediumName* Medium name  
*libraryName* Library name  
*substanceName* Substance name

**8.3.2.4 EXPORT double TwoPhaseMedium\_getCriticalTemperature\_ (const char \* *mediumName*, const char \* *libraryName*, const char \* *substanceName*)**

Get critical temperature.

This function returns the critical temperature of the specified medium.

**Parameters:**

*mediumName* Medium name  
*libraryName* Library name  
*substanceName* Substance name

**8.3.2.5** EXPORT double TwoPhaseMedium\_getMolarMass\_ (const char \* *mediumName*, const char \* *libraryName*, const char \* *substanceName*)

Get molar mass.

This function returns the molar mass of the specified medium.

**Parameters:**

*mediumName* Medium name

*libraryName* Library name

*substanceName* Substance name

**8.3.2.6** EXPORT void TwoPhaseMedium\_setBubbleState\_ (int *uniqueID*, int *phase*, int \* *state\_uniqueID*, int \* *state\_phase*, const char \* *mediumName*, const char \* *libraryName*, const char \* *substanceName*)

Compute bubble state.

This function computes the bubble state for the specified medium.

**Parameters:**

*uniqueID* Unique ID number

*phase* Phase (2 for two-phase, 1 for one-phase, 0 if not known)

*state\_uniqueID* Pointer to return unique ID number for state record

*state\_phase* Pointer to return phase for state record

*mediumName* Medium name

*libraryName* Library name

*substanceName* Substance name

**8.3.2.7** EXPORT void TwoPhaseMedium\_setDewState\_ (int *uniqueID*, int *phase*, int \* *state\_uniqueID*, int \* *state\_phase*, const char \* *mediumName*, const char \* *libraryName*, const char \* *substanceName*)

Compute dew state.

This function computes the dew state for the specified medium.

**Parameters:**

*uniqueID* Unique ID number

*phase* Phase (2 for two-phase, 1 for one-phase, 0 if not known)

*state\_uniqueID* Pointer to return unique ID number for state record

*state\_phase* Pointer to return phase for state record

*mediumName* Medium name

*libraryName* Library name

*substanceName* Substance name

**8.3.2.8** EXPORT void TwoPhaseMedium\_setSat\_p\_ (double *p*, int *uniqueID*, double \* *sat\_psat*, double \* *sat\_Tsat*, int \* *sat\_uniqueID*, const char \* *mediumName*, const char \* *libraryName*, const char \* *substanceName*)

Compute saturation properties from *p*.

This function computes the saturation properties for the specified inputs. If the function is called with *uniqueID* == 0 a new transient unique ID is assigned and the function is called again with this new transient unique ID number.

**Parameters:**

*p* Pressure  
*uniqueID* Unique ID number  
*sat\_psat* Pointer to return pressure for saturation record  
*sat\_Tsat* Pointer to return temperature for saturation record  
*sat\_uniqueID* Pointer to return unique ID number for saturation record  
*mediumName* Medium name  
*libraryName* Library name  
*substanceName* Substance name

**8.3.2.9** EXPORT void TwoPhaseMedium\_setSat\_p\_state\_ (int *uniqueID*, double \* *sat\_psat*, double \* *sat\_Tsat*, int \* *sat\_uniqueID*)

Compute saturation properties from within BaseProperties.

This function computes the saturation properties and is designed to be called from within the BaseProperties model. The saturation properties are set according to the medium pressure

**Parameters:**

*uniqueID* Unique ID number  
*sat\_psat* Pointer to return pressure for saturation record  
*sat\_Tsat* Pointer to return temperature for saturation record  
*sat\_uniqueID* Pointer to return unique ID number for saturation record

**8.3.2.10** EXPORT void TwoPhaseMedium\_setSat\_T\_ (double *T*, int *uniqueID*, double \* *sat\_psat*, double \* *sat\_Tsat*, int \* *sat\_uniqueID*, const char \* *mediumName*, const char \* *libraryName*, const char \* *substanceName*)

Compute saturation properties from *T*.

This function computes the saturation properties for the specified inputs. If the function is called with *uniqueID* == 0 a new transient unique ID is assigned and the function is called again with this new transient unique ID number.

**Parameters:**

*T* Temperature  
*uniqueID* Unique ID number  
*sat\_psat* Pointer to return pressure for saturation record

*sat\_Tsat* Pointer to return temperature for saturation record  
*sat\_uniqueID* Pointer to return unique ID number for saturation record  
*mediumName* Medium name  
*libraryName* Library name  
*substanceName* Substance name

**8.3.2.11** EXPORT void TwoPhaseMedium\_setState\_dT\_ (double *d*, double *T*, int *phase*, int *uniqueID*, int \* *state\_uniqueID*, int \* *state\_phase*, double \* *state\_d*, double \* *state\_h*, double \* *state\_p*, double \* *state\_s*, double \* *state\_T*, const char \* *mediumName*, const char \* *libraryName*, const char \* *substanceName*)

Compute properties from *d*, *T*, and *phase*.

This function computes the properties for the specified inputs. If the function is called with *uniqueID* == 0 a new transient unique ID is assigned and the function is called again with this new transient unique ID number.

**Parameters:**

*d* Density  
*T* Temperature  
*phase* Phase (2 for two-phase, 1 for one-phase, 0 if not known)  
*uniqueID* Unique ID number  
*state\_uniqueID* Pointer to return unique ID number for state record  
*state\_phase* Pointer to return phase for state record  
*state\_d* Pointer to return density for state record  
*state\_h* Pointer to return specific enthalpy for state record  
*state\_p* Pointer to return pressure for state record  
*state\_s* Pointer to return specific entropy for state record  
*state\_T* Pointer to return temperature for state record  
*mediumName* Medium name  
*libraryName* Library name  
*substanceName* Substance name

**8.3.2.12** EXPORT void TwoPhaseMedium\_setState\_ph\_ (double *p*, double *h*, int *phase*, int *uniqueID*, int \* *state\_uniqueID*, int \* *state\_phase*, double \* *state\_d*, double \* *state\_h*, double \* *state\_p*, double \* *state\_s*, double \* *state\_T*, const char \* *mediumName*, const char \* *libraryName*, const char \* *substanceName*)

Compute properties from *p*, *h*, and *phase*.

This function computes the properties for the specified inputs. If the function is called with *uniqueID* == 0 a new transient unique ID is assigned and the function is called again with this new transient unique ID number.



**Parameters:**

*p* Pressure  
*h* Specific enthalpy  
*phase* Phase (2 for two-phase, 1 for one-phase, 0 if not known)  
*uniqueID* Unique ID number  
*state\_uniqueID* Pointer to return unique ID number for state record  
*state\_phase* Pointer to return phase for state record  
*state\_d* Pointer to return density for state record  
*state\_h* Pointer to return specific enthalpy for state record  
*state\_p* Pointer to return pressure for state record  
*state\_s* Pointer to return specific entropy for state record  
*state\_T* Pointer to return temperature for state record  
*mediumName* Medium name  
*libraryName* Library name  
*substanceName* Substance name

**8.3.2.13** EXPORT void TwoPhaseMedium\_setState\_ps\_ (double *p*, double *s*, int *phase*, int *uniqueID*, int \* *state\_uniqueID*, int \* *state\_phase*, double \* *state\_d*, double \* *state\_h*, double \* *state\_p*, double \* *state\_s*, double \* *state\_T*, const char \* *mediumName*, const char \* *libraryName*, const char \* *substanceName*)

Compute properties from *p*, *s*, and *phase*.

This function computes the properties for the specified inputs. If the function is called with *uniqueID* == 0 a new transient unique ID is assigned and the function is called again with this new transient unique ID number.

**Parameters:**

*p* Pressure  
*s* Specific entropy  
*phase* Phase (2 for two-phase, 1 for one-phase, 0 if not known)  
*uniqueID* Unique ID number  
*state\_uniqueID* Pointer to return unique ID number for state record  
*state\_phase* Pointer to return phase for state record  
*state\_d* Pointer to return density for state record  
*state\_h* Pointer to return specific enthalpy for state record  
*state\_p* Pointer to return pressure for state record  
*state\_s* Pointer to return specific entropy for state record  
*state\_T* Pointer to return temperature for state record  
*mediumName* Medium name  
*libraryName* Library name  
*substanceName* Substance name

**8.3.2.14** EXPORT void TwoPhaseMedium\_setState\_pT\_ (double *p*, double *T*, int *phase*, int *uniqueID*, int \* *state\_uniqueID*, int \* *state\_phase*, double \* *state\_d*, double \* *state\_h*, double \* *state\_p*, double \* *state\_s*, double \* *state\_T*, const char \* *mediumName*, const char \* *libraryName*, const char \* *substanceName*)

Compute properties from *p*, *T*, and *phase*.

This function computes the properties for the specified inputs. If the function is called with *uniqueID* == 0 a new transient unique ID is assigned and the function is called again with this new transient unique ID number.

Attention: The phase input is ignored for this function!

**Parameters:**

*p* Pressure

*T* Temperature

*phase* Phase (2 for two-phase, 1 for one-phase, 0 if not known)

*uniqueID* Unique ID number

*state\_uniqueID* Pointer to return unique ID number for state record

*state\_phase* Pointer to return phase for state record

*state\_d* Pointer to return density for state record

*state\_h* Pointer to return specific enthalpy for state record

*state\_p* Pointer to return pressure for state record

*state\_s* Pointer to return specific entropy for state record

*state\_T* Pointer to return temperature for state record

*mediumName* Medium name

*libraryName* Library name

*substanceName* Substance name

## 8.4 include.h File Reference

generic include file

```
#include <map>
#include <string>
#include "errorhandling.h"
```

### Defines

- `#define DYMOLA 1`  
*Modelica compiler is Dymola.*
- `#define OPEN_MODELICA 0`  
*Modelica compiler is OpenModelica.*
- `#define FLUIDPROP 0`  
*FluidProp solver.*
- `#define BUILD_DLL 0`  
*Build project into a DLL.*
- `#define BUILD_LIB 1`  
*Build project into a LIB.*
- `#define MAX_TRANSIENT_MEDIUM 1000`  
*Maximum number of non-overlapping transient medium objects.*
- `#define NAN 0xffffffff`  
*Not a number.*

### 8.4.1 Detailed Description

generic include file

This is a generic include for for the entire ExternalMediaPackage project. It defines some important preprocessor variables that might have to be changed by the user.

Uncomment the define directives as appropriate

Francesco Casella, Christoph Richter, Oct 2006 Copyright Politecnico di Milano and TU Braunschweig

### 8.4.2 Define Documentation

#### 8.4.2.1 `#define BUILD_DLL 0`

Build project into a DLL.

Set this preprocessor variable to 1 if the project is built into a dynamic link library. This setting influences the error reporting mechanism as well as the export statement.

See also:

**BUILD\_LIB** (p. 72)

#### 8.4.2.2 **#define BUILD\_LIB 1**

Build project into a LIB.

Set this preprocessor variable to 1 if the project is built into a static library. This setting influences the error reporting mechanism as well as the export statement.

See also:

**BUILD\_DLL** (p. 71)

#### 8.4.2.3 **#define DYMOLA 1**

Modelica compiler is Dymola.

Set this preprocessor variable to 1 if Dymola is the Modelica compiler that is going to be used with the compiled library.

See also:

**OPEN\_MODELICA** (p. 72)

#### 8.4.2.4 **#define FLUIDPROP 0**

FluidProp solver.

Set this preprocessor variable to 1 to include the interface to the FluidProp solver developed and maintained by Francesco Casella.

#### 8.4.2.5 **#define MAX\_TRANSIENT\_MEDIUM 1000**

Maximum number of non-overlapping transient medium objects.

Increase this number if you ran a large model with more than a thousand instances of ThermodynamicState and SaturationState outside BaseProperties records, and without explicit uniqueID handling

#### 8.4.2.6 **#define NAN 0xffffffff**

Not a number.

This value is used as not a number value. It can be changed by the user if there is a more appropriate value.

#### 8.4.2.7 **#define OPEN\_MODELICA 0**

Modelica compiler is OpenModelica.

Set this preprocessor variable to 1 if OpanModelica is the Modelica compiler that is going to be used with the compiled library.

See also:

[DYMOLA](#) (p. 72)

# Index

- ~BaseSolver
  - BaseSolver, 17
- ~BaseTwoPhaseMedium
  - BaseTwoPhaseMedium, 27
- ~TwoPhaseMedium
  - TwoPhaseMedium, 42
- addMedium
  - MediumMap, 33
- addSolverMedium
  - MediumMap, 33
- addTransientMedium
  - MediumMap, 33
- BaseSolver, 15
  - ~BaseSolver, 17
  - BaseSolver, 17
  - computeDerivatives, 21
  - isentropicEnthalpy, 21
  - setBubbleState, 20
  - setDewState, 20
  - setFluidConstants, 18
  - setSat\_p, 18
  - setSat\_p\_state, 18
  - setSat\_T, 18
  - setState\_dT, 19
  - setState\_ph, 19
  - setState\_ps, 19
  - setState\_pT, 20
- BaseTwoPhaseMedium, 22
  - ~BaseTwoPhaseMedium, 27
  - BaseTwoPhaseMedium, 26
  - getBubbleUniqueID, 29
  - getDewUniqueID, 29
  - reinitMedium, 27
  - setBubbleState, 29
  - setDewState, 29
  - setSat\_p, 27
  - setSat\_p\_state, 28
  - setSat\_T, 28
  - setSolver, 27
  - setState\_dT, 28
  - setState\_ph, 28
  - setState\_ps, 28
  - setState\_pT, 29
- BUILD\_DLL
  - include.h, 71
- BUILD\_LIB
  - include.h, 72
- changeMedium
  - MediumMap, 34
- computeDerivatives
  - BaseSolver, 21
- deleteMedium
  - MediumMap, 34
- DYMOLA
  - include.h, 72
- errorhandling.h, 47
  - errorMessage, 47
  - warningMessage, 48
- errorMessage
  - errorhandling.h, 47
- externalmedialib.cpp, 49
  - TwoPhaseMedium\_createMedium\_, 54
  - TwoPhaseMedium\_-
    - getCriticalMolarVolume\_, 54
  - TwoPhaseMedium\_getCriticalPressure\_, 54
  - TwoPhaseMedium\_-
    - getCriticalTemperature\_, 54
  - TwoPhaseMedium\_getMolarMass\_, 55
  - TwoPhaseMedium\_setBubbleState\_, 55
  - TwoPhaseMedium\_setDewState\_, 55
  - TwoPhaseMedium\_setSat\_p\_, 56
  - TwoPhaseMedium\_setSat\_p\_state\_, 56
  - TwoPhaseMedium\_setSat\_T\_, 56
  - TwoPhaseMedium\_setState\_dT\_, 57
  - TwoPhaseMedium\_setState\_ph\_, 57
  - TwoPhaseMedium\_setState\_ps\_, 58
  - TwoPhaseMedium\_setState\_pT\_, 58
  - TwoPhaseMedium\_setStateDefault\_, 59
- externalmedialib.h, 60
  - TwoPhaseMedium\_createMedium\_, 64
  - TwoPhaseMedium\_-
    - getCriticalMolarVolume\_, 65
  - TwoPhaseMedium\_getCriticalPressure\_, 65

- TwoPhaseMedium\_
  - getCriticalTemperature\_, 65
  - TwoPhaseMedium\_getMolarMass\_, 65
  - TwoPhaseMedium\_setBubbleState\_, 66
  - TwoPhaseMedium\_setDewState\_, 66
  - TwoPhaseMedium\_setSat\_p\_, 66
  - TwoPhaseMedium\_setSat\_p\_state\_, 67
  - TwoPhaseMedium\_setSat\_T\_, 67
  - TwoPhaseMedium\_setState\_dT\_, 68
  - TwoPhaseMedium\_setState\_ph\_, 68
  - TwoPhaseMedium\_setState\_ps\_, 69
  - TwoPhaseMedium\_setState\_pT\_, 69
- FluidConstants, 31
- FLUIDPROP
  - include.h, 72
- getBubbleUniqueID
  - BaseTwoPhaseMedium, 29
- getDewUniqueID
  - BaseTwoPhaseMedium, 29
- getSolver
  - SolverMap, 36
- include.h, 71
  - BUILD\_DLL, 71
  - BUILD\_LIB, 72
  - DYMOLA, 72
  - FLUIDPROP, 72
  - MAX\_TRANSIENT\_MEDIUM, 72
  - NAN, 72
  - OPEN\_MODELICA, 72
- initializeFields
  - TwoPhaseMediumProperties, 46
- isentropicEnthalpy
  - BaseSolver, 21
- MAX\_TRANSIENT\_MEDIUM
  - include.h, 72
- medium
  - MediumMap, 34
- MediumMap, 32
  - addMedium, 33
  - addSolverMedium, 33
  - addTransientMedium, 33
  - changeMedium, 34
  - deleteMedium, 34
  - medium, 34
  - solverMedium, 34
- NAN
  - include.h, 72
- OPEN\_MODELICA
  - include.h, 72
- phase
  - TwoPhaseMediumProperties, 46
- reinitMedium
  - BaseTwoPhaseMedium, 27
- setBubbleState
  - BaseSolver, 20
  - BaseTwoPhaseMedium, 29
- setDewState
  - BaseSolver, 20
  - BaseTwoPhaseMedium, 29
- setFluidConstants
  - BaseSolver, 18
  - TestSolver, 39
- setSat\_p
  - BaseSolver, 18
  - BaseTwoPhaseMedium, 27
  - TestSolver, 39
- setSat\_p\_state
  - BaseSolver, 18
  - BaseTwoPhaseMedium, 28
  - TestSolver, 40
- setSat\_T
  - BaseSolver, 18
  - BaseTwoPhaseMedium, 28
  - TestSolver, 39
- setSolver
  - BaseTwoPhaseMedium, 27
- setState\_dT
  - BaseSolver, 19
  - BaseTwoPhaseMedium, 28
  - TestSolver, 40
- setState\_ph
  - BaseSolver, 19
  - BaseTwoPhaseMedium, 28
  - TestSolver, 40
- setState\_ps
  - BaseSolver, 19
  - BaseTwoPhaseMedium, 28
  - TestSolver, 41
- setState\_pT
  - BaseSolver, 20
  - BaseTwoPhaseMedium, 29
  - TestSolver, 41
- solverKey
  - SolverMap, 36
- SolverMap, 36
  - getSolver, 36
  - solverKey, 36
- solverMedium
  - MediumMap, 34
- std, 13

- TestSolver, 38
  - setFluidConstants, 39
  - setSat\_p, 39
  - setSat\_p\_state, 40
  - setSat\_T, 39
  - setState\_dT, 40
  - setState\_ph, 40
  - setState\_ps, 41
  - setState\_pT, 41
- TwoPhaseMedium, 42
  - ~TwoPhaseMedium, 42
  - TwoPhaseMedium, 42
- TwoPhaseMedium\_createMedium\_
  - externalmedialib.cpp, 54
  - externalmedialib.h, 64
- TwoPhaseMedium\_getCriticalMolarVolume\_
  - externalmedialib.cpp, 54
  - externalmedialib.h, 65
- TwoPhaseMedium\_getCriticalPressure\_
  - externalmedialib.cpp, 54
  - externalmedialib.h, 65
- TwoPhaseMedium\_getCriticalTemperature\_
  - externalmedialib.cpp, 54
  - externalmedialib.h, 65
- TwoPhaseMedium\_getMolarMass\_
  - externalmedialib.cpp, 55
  - externalmedialib.h, 65
- TwoPhaseMedium\_setBubbleState\_
  - externalmedialib.cpp, 55
  - externalmedialib.h, 66
- TwoPhaseMedium\_setDewState\_
  - externalmedialib.cpp, 55
  - externalmedialib.h, 66
- TwoPhaseMedium\_setSat\_p\_
  - externalmedialib.cpp, 56
  - externalmedialib.h, 66
- TwoPhaseMedium\_setSat\_p\_state\_
  - externalmedialib.cpp, 56
  - externalmedialib.h, 67
- TwoPhaseMedium\_setSat\_T\_
  - externalmedialib.cpp, 56
  - externalmedialib.h, 67
- TwoPhaseMedium\_setState\_dT\_
  - externalmedialib.cpp, 57
  - externalmedialib.h, 68
- TwoPhaseMedium\_setState\_ph\_
  - externalmedialib.cpp, 57
  - externalmedialib.h, 68
- TwoPhaseMedium\_setState\_ps\_
  - externalmedialib.cpp, 58
  - externalmedialib.h, 69
- TwoPhaseMedium\_setState\_pT\_
  - externalmedialib.cpp, 58
  - externalmedialib.h, 69
- TwoPhaseMedium\_setStateDefault\_
  - externalmedialib.cpp, 59
- TwoPhaseMediumProperties, 44
  - initializeFields, 46
  - phase, 46
  - TwoPhaseMediumProperties, 46
- warningMessage
  - errorhandling.h, 48