Aspen Custom Modeler 2004.1

Modeling Language Reference Guide

aspentech
driving process profitability

Who Should Read this Guide

This guide contains detailed reference information about the modeling language, including syntax details and examples.

Contents

Introducing Aspen Custom Modeler

Aspen Custom Modeler (ACM) is an easy-to-use tool for creating, editing and re-using models of process units. You build simulation applications by combining these models on a graphical flowsheet. Models can use inheritance and hierarchy and can be re-used directly or built into libraries for distribution and use. Dynamic, steady-state, parameter estimation and optimization simulations are solved in an equation-based manner which provides flexibility and power.

ACM uses an object-oriented modeling language, editors for icons and tasks, and Microsoft Visual Basic for scripts. ACM is customizable and has extensive automation features, making it simple to combine with other products such as Microsoft Excel and Visual Basic. This allows you to build complete applications for non-experts to use.

1 Modeling Language Concepts

The Aspen Custom Modeler modeling language uses a number of common concepts throughout. This chapter describes the following concepts:

- Statements
- Type definitions
- Declaration statements
- Assignments
- Assigning array elements
- Assignment lists
- Assignments using WITHIN and ENDWITH
- Naming conventions
- Duplicate names: BLOCK and STREAM
- Parameters
- Sets
- Set arithmetic
- Arrays
- Conversions of atomic types
- Inheritance
- Type relatives
- Component Lists
- Built-in type definitions

Statements

Use statements to make assignments of values, define equations, declare items such as variables or models, and make connections.

Statements include one of the syntax elements shown in the following table.

Syntax Element	Type of Statement
AS	Declaration
:	Value assignment

=	Equation
CONNECT AND	Connection
LINK AND	Link

Statements must end with the ; (semi-colon).

Statement Examples

Create an instance of a variable in a model:

```
T AS Temperature;
```

Assign a value to a variable and specify that the variable value is known:

```
Feed.T: 75.0, Fixed;
```

Equate two variables in a model:

```
FlowIn = FlowOut;
```

Make a connection between the input and output of two blocks:

```
CONNECT Pump1.Output1 AND Valve2.Input1;
```

Make a connection between the port of a sub-model and the port of the containing model:

```
LINK Tank5.Output1 AND MultiOutput1;
```

Type Definitions

Type definitions introduce and describe new types for models, streams, variables, structures, procedures, ports and parameters.

For further information on types, see Modeling Language for Types. Chapter 3.

Type Definition Syntax

The syntax for all type definitions is of the form:

```
TypeKeyword TypeName  USES UsedTypeName

   DefinitionBody
```

```
END
```

TypeKeyword	One of these keywords: MODEL, STREAM, VARIABLE, STRUCTURE, PROCEDURE, PORT, PARAMETER
TypeName	The name of the type being defined. This name must be unique within the scope in which the definition is made.
UsedTypeName	Optional - The name of a type (which has been previously defined), from which this type inherits properties
DefinitionBody	The content of the definition. This varies depending on the object that is defined. The definition body can be one of the types listed in Chapter 3. In the simplest case, the definition body is just a list of the declarations or assignments of the type's properties.

Type Definition Example

This example defines the new variable type Temperature, and assigns default values for all variables that use this type.

```
VARIABLE Temperature
  Value: 1.0;
  Lower: -40.0;
  Upper: 300.0;
  Scale: 10.0;
  Description: "Temperature";
  PhysicalQuantity: "Temperature";
END
```

Declaration Statements

A declaration statement declares a property by giving it a name and defining its type. The syntax for all declarations has the following form:

```
Identifier, Identifier ... AS FlagList TypeName
(AssignmentList);
```

Identifier	The name of a property being declared. This can be a list if more than one properties are to be declared with the same FlagList,

	TypeName and AssignmentList.
FlagList	Optionlal list of qualifiers, separated by commas, that apply to the property. These qualifiers affect the implementation of the property, and are optional. For example, the direction of a port is defined in the FlagList.
TypeName.....	The name of the type of which the property is an instance.
AssignmentList	Optional comma-separated list of assignments of values to properties of the property. The assignments in the list are of the form PropertyName:Value. If the type of the value unambiguously identifies the property, you can omit PropertyName.

Declaration Statement Remarks

Note that not all declarations are valid in all contexts. For example, you can declare a port within a model definition, but not within a variable type definition.

Declaration Statement Example

Below are some example declaration statements that could be found within a model.

```
P AS Pressure (fixed, Description: ”Inlet Pressure”);
```

```
OUT_V as output MaterialPort
```

P is declared as a variable of type pressure, with a specification of fixed, and a textual description. OUT_V is declared as a port of type MaterialPort, and the flag output identifies it as an output port.

Assignments

Assignments are used to change the value of a property of an object, such as a variable or a parameter.

Assignment Syntax

The modeling language uses a **:** (colon) as its assignment operator:

```
Object.Property: ValueExpression;
```

Object	A variable or parameter
Property.....	A property of the variable or parameter, for example Value, Spec, Description etc. **Note:** If you do not specify a property, Value is assumed.
ValueExpression	An expression for the property

value

Assignment Remarks

Assignments to expressions are treated as constraints by the compiler. This means that the expression is re-evaluated whenever the value of any term within it changes. The result is then automatically assigned to the property.

You can assign values to properties of variables or parameters in several places, for example in models, flowsheet constraints and interactively. From highest to lowest, the order of precedence in which variable and parameter properties are changed is:

- Changes made in the following ways are made immediately and overwrite any previous assignment:
  - Interactive changes, for example, using a table, using the automation interface, running a script, or applying a snapshot with Use
  - Values calculated as the result of running a simulation
- Assignments written into the Flowsheet Constraints section take precedence over assignments in models.
- If an assignment in a model uses an expression, and one of the values in the expression changes, the expression is re-evaluated and the assigned value is updated. However if the value has previously been changed by a higher precedence operation, this updating is disabled to avoid overriding this change.
- If a block uses a model that uses hierarchy, assignments made in the parent model take precedence over those in child models in the hierarchy.
- If a block uses a model that uses inheritance, assignments in the derived model take precedence over those in the inherited model.
- If properties are not assigned anywhere else, they will remain at the default values defined for their variable type or parameter type.

Assignment Examples

Assign the value 5 to the Value property of the variable x:

```
x: 5.0;
```

Assign the value "Cylinder" to the Value property of the string parameter Geometry:

```
Geometry: "Cylinder";
```

Assign the value True to the Value property of the logical parameter PressureDriven:

```
PressureDriven: True;
```

Assign a value to the variable Lag using an expression, and assign a value to Lag2 that is dependant on the value of Lag:

```
Lag: 5/60;
Lag2: Lag/2;
```

Assign values for the Upper and Lower bounds of variable x:

```
x.Lower: -10;
x.Upper: 20;
```

For further examples of assignments, see Defining Specifications in Models Examples and Defining Specifications in Flowsheets Examples.

Assignments to Array Elements

Assignments to an array variable will assign the specified value to all members of the array. However you can assign values to individual array elements.

Assigning to Array Elements Syntax

```
ArrayVariable(Index) :  Value ;
```

ArrayVariable.....	The name of the array variable
Index.....	The index of the array. This must be a member of the index set for the array
Value.....	The value to be assigned to the array element

Assigning to Array Elements Example

The following example shows the values of an array can be assigned individually:

```
Components AS StringSet(["H2O","H2+","Zn++","Ca++"]);
mw(Components) AS MoleWeight;
mw("H2O"): 18;
mw("Ca++"): 40.1;
```

Note: For convenience Components is defined as the index set for the array mw. The array variable mw in the above example could have been defined directly with the statement:

```
mw(["H2O","H2+","Zn++","Ca++"]) AS MoleWeight;
```

The [] characters show the start and end of the list of elements in the set.

Assignment Lists

You can use assignment lists in variable assignments to avoid repeating path names.

Assignment List Syntax

```
Property :  Assignment, Assignment, ...  ;
```

Property	Property, such as a variable within a block
Assignment	Value assigned to the Property

Assignment List Example

The following example assigns 25 to the value attribute and Fixed to the specification attribute of the variable Tank.Q.

```
Tank.Q: VALUE:25.0, SPEC: Fixed, LOWER: -10.0;
```

If the property name is omitted it is assumed that the assignment is to the value or spec attribute. This example can be shortened to:

```
Tank.Q: 25.0, FIXED, LOWER: -10.0;
```

Assignments using WITHIN and ENDWITH

If you want to make several assignments within the same scope, you can reduce the amount of typing by using the WITHIN and ENDWITH keywords.

WITHIN and ENDWITH Syntax

```
WITHIN Scope
   AssignmentList
ENDWITH
```

Scope	The block or variable to which subsequent assignments apply
AssignmentList	The list of assignments

WITHIN / ENDWITH Remarks

You can use WITHIN / ENDWITH statements within a model when making assignments to variables in a sub-model.

WITHIN / ENDWITH Example

Assign values to multiple variables within a sub-model used within a model.

```
Rxn1 AS PowerLawRxn;
WITHIN Rxn1
  PreExp: 1.45E6;
  EAct  : 2374.7;
  TExp  : 1.2;
ENDWITH
```

Alternatively, you could type each specification explicitly:

```
Rxn1 AS PowerLawRxn;
Rxn1.PreExp: 1.45E6;
Rxn1.EAct  : 2374.7;
Rxn1.TExp  : 1.2;
```

Naming Conventions

You use names when you:

- Define a type, such as a model or a stream type.
- Add a property to a type, such as a variable.

Important Note: You cannot create a type which has the same name as a built-in type or a keyword.

Make sure that the names you choose:

- Are unique within their scope, for example variable names must be unique within a model, but two different models can have a variable with the same name.
- Include only alphanumeric characters and the **_** (underscore) character.
- Start with an alphabetic character.
- Do not begin with **am_**.

Because you may use names to identify objects from external programs such as Microsoft Visual Basic or Microsoft Excel, you should apply the naming rules for Visual Basic.

Duplicate Names: BLOCK and STREAM

Within a given scope, the names you use for all properties must be unique. The one exception to this is that you can have a block and a stream with the same name within your flowsheet. This feature is for compatibility with other AspenTech products such as Aspen Plus. In general it is still best to avoid using the same name for a block and a stream.

If you use the same name for a block and a stream you must refer to each using the following syntax:

Block Syntax

```
BLOCKS("BlockName")
```

Stream Syntax

```
STREAMS("StreamName")
```

BlockName	Name of a block in the flowsheet
StreamName	Name of a stream in the flowsheet

You can use the same syntax in both input files and in Microsoft Visual Basic.

BLOCK Keyword Example

The following is a global equation which equates two variables in a block and a stream which have the same name:

```
BLOCKS("B1").Tin = STREAMS("B1").T;
```

Parameters

Use parameters to define values that are fixed for the whole simulation, such as acceleration due to gravity, or the gas constant. You can also use parameters for fixed physical quantities such as the number of trays in a column. Like variables, parameters are declared within types such as models or stream types.

Parameters Syntax

```
ParameterName AS ParameterType  (Value);
```

ParameterName	Name of the parameter
ParameterType	Can be IntegerParameter, RealParameter, StringParameter, LogicalParameter, or a user-defined parameter type

Value.....	Optional value assigned to the parameter

Parameters Remarks

The key difference between variables and parameters is that variables can be solved for during a simulation run, whereas parameters always have a known value. Also parameters can be used in conditional statements or loops within models to determine which variables and equations will be active for a particular instance of a model.

Parameters may be of one of the built in types IntegerParameter, StringParameter, LogicalParameter or RealParameter., or of a user defined type defined using the PARAMETER keyword.

Parameters Examples

Define a parameter for the gravitational constant g:

```
g AS RealParameter (9.81);
```

In the next example the model Column has a parameter NStages which defines the number of stages. This parameter is used to define an array of Tray models called Stages. NSTages is given a default value of 10, but this can be changed when the Column model is instanced as a block in the flowsheet.

```
MODEL Column
  NStages AS IntegerParameter(10);
  Stages([1:NStages]) AS Tray;
.
.
END
```

Sets

Sets are most commonly used to index array variables. For information on arrays, see Arrays.

A set is a collection of objects that has no duplicate values and is not in any specific order. You can manipulate sets using set functions, such as union, intersection, and difference.

Sets Syntax

```
SetName AS SetType (SetContents)  ;
```

SetName	The set name
SetType.....	Can be IntegerSet or StringSet
SetContents	An optional list of the contents of

the set

Sets Remarks

You can define sets that contain either integers (IntegerSet) or strings (StringSet).

An IntegerSet is a set whose members are integers, for example [1,2,5].

A StringSet is a set whose members are strings, for example [“Methane”, “Ethane”, “Propane”].

You can define the set contents at the same time as you define the set, or you can define the set contents later. The set contents can be defined by a list of objects enclosed between **[** and **]** (brackets), or by a set expression involving previously defined sets.

Sets Examples

The set MySet is defined as containing objects that are of integer values. The contents of the set are not yet defined.

```
MySet AS IntegerSet;
```

MySet is defined as containing integer objects, and the set is immediately defined as an empty set. This set has no contents:

```
MySet AS IntegerSet([]);
```

MySet is defined as containing four integers 1, 2, 3, and 4:

```
MySet AS IntegerSet([1, 2, 3, 4]);
```

MySet is defined as containing four integers 1, 2, 3, and 4 using a colon to define a range of numbers:

```
MySet  AS IntegerSet([1:4]);
```

MySet is defined as a set containing the strings "Yes" and "No":

```
MySet AS StringSet(["Yes", "No"]);
```

Set Arithmetic

The following set arithmetic functions are available:

Function name	Symbol	Description
UNION	+	The UNION of two sets is the set of elements that are in both sets but without duplicates.
DIFFERENCE	–	The DIFFERENCE of two sets is the set of elements in the first set but not in the second set.

INTERSECTION	*	The INTERSECTION of two sets is the set of elements that are common to both sets.
SYMDIFF	none	The SYMDIFF of two sets is the set of the elements that are in one set but not in the other. This can be defined as the difference between the union and the intersection of the two sets.

Set Arithmetic Function Syntax

You can use the symbol or the function name for set arithmetic.

```
SetExpression  SetFunctionSymbol  SetExpression;
```

SetExpression ..... A set expression, such as the name of a set, or a set definition enclosed by **[ ]** (brackets)

SetFunctionSymbol ..... One of the set function symbols: + - *

or

```
SetFunctionName (SetExpression1, SetExpression2, ... );
```

SetFunctionName..... One of the set function names, such as UNION, INTERSECTION, etc.

SetExpression ..... A set expression, such as the name of a set, or a set definition enclosed by **[ ]** (brackets)

Set Arithmetic Example

Define SetA and SetB:

```
SetA AS IntegerSet([1, 2, 3, 4]);
SetB AS IntegerSet([3, 4, 5, 6]);
```

The union of SetA and SetB is [1, 2, 3, 4, 5, 6]:

```
SetA + SetB
```

or

```
UNION (SetA, SetB);
```

The intersection of SetA and SetB is [3, 4]:

```
SetA * SetB
```

or

```
INTERSECTION (SetA, SetB);
```

The difference of SetA and SetB is [1, 2]:

```
SetA - SetB
```

or

```
DIFFERENCE (SetA, SetB) ;
```

The symmetric difference of SetA and SetB is [1, 2, 5, 6]:

```
SYMDIFF (SetA, SetB)
```

Arrays

An array is a collection of objects indexed by either one integer set or one string set for each dimension of the array.

You can use arrays to define a collection of variables, parameters, ports, model instances, or streams.

Note: Global arrays are not supported.

Arrays Remarks

Because an array is indexed by a set, it is possible for the index of the array to not take consecutive integer values. You can define an array that is indexed by an integer set that has some of the integer numbers "missing". This is useful for modeling trays in a distillation column, where, for example, you may want to use a different model for a feed tray.

As arrays can be indexed by a string set, you can define an array indexed by component names. For more information, see Chapter 3, Using the ComponentList Property.

Arrays Examples

The following line defines an array variable. The variable SectionTemp is indexed explicitly over the integer set from 1 to 10:

```
SectionTemp([1:10]) AS Temperature;
```

The variable FeedMolFrac is indexed by the set Components, which contains three component names:

```
Components AS StringSet(["N2", "O2", "CO2"]);

FeedMolFrac(Components) AS MoleFraction;
```

The variable SectionMolFrac is defined as a two dimensional array. The first dimension is defined as the set Components and the second dimension is the set of integers from 1 to 10:

```
SectionMolFrac(Components,[1:10]) AS MolFraction;
```

The following example shows how to define an array of model instances:

```
MODEL Tray
  .
  .
END

MODEL FeedTray
  .
  .
END

MODEL  Column
  NTrays    AS IntegerParameter(10);
  NFeedTray AS IntegerParameter(5);

  Stage([1:NTrays] - NFeedTray) AS Tray;
  FeedStage AS FeedTray;
  .
  .
END
```

The model Column uses the previously defined models Tray and FeedTray. The parameter NTrays is the total number of trays and NFeedTray is the tray number of the feed tray. NTray is used to define an array of instances of the Tray model. Set arithmetic is used to create an array of Trays called Stage. Stage(5) is excluded from this array because this is the feed stage, which is represented by a different model, FeedTray..

The definition for Stage is:

```
Stage([1, 2, 3, 4, 6, 7, 8, 9, 10])
```

Atomic Type Conversions

The CONVERTTO operator enables you to convert a string or the first element in a set of strings to the integer value that it represents.

You can also use CONVERTTO to convert a REAL, INTEGER, or LOGICAL value to a string. For example, you could use CONVERTTO in a task, to print out variable values (because the PRINT statement in a task allows only strings).

CONVERTTO Syntax

Use the following syntax for CONVERTTO:

```
CONVERTTO '('ResultType', 'Expression')';
```

ResultType	Can be INTEGER or STRING
Expression	If *ResultType* is INTEGER, *Expression* must be of type STRING or STRINGSET If *ResultType* is STRING, *Expression* must be of type REAL, INTEGER, or LOGICAL

Example of Using CONVERTTO

The following example shows how to use CONVERTTO to convert to integer values:

```
x_string as stringparameter ("5");
x_int    as IntegerParameter(CONVERTTO(INTEGER, "3"));
//initializes x_int with the value 3
x_int: CONVERTTO(INTEGER, x_string); //assigns the integer
value of x_string (5) to x_int
```

The following example shows how to use CONVERTTO to convert to a string:

```
task MyTask
  s2 as StringParameter;
  s2: CONVERTTO(string, V); // V is a variable accessible
to the task
  print "value of V is " + s2;
end
```

Note that CONVERTTO cannot be used directly in a print statement.

Inheritance

Inheritance enables you to create new types which build upon existing ones. You describe an inheritance relationship between two type definitions with the keyword USES.

Inheritance Syntax

```
Type TypeName USES TypeName
  Statements
END
```

Type	One of the supported types, such as Model or Variable
TypeName	The name of the new type being defined
Statements	Statements which extend the definition of the inherited type

Inheritance Remarks

If you define a type using inheritance, the type inherits all the properties of the type from which it inherits. You can then extend the type definition with statements that add extra properties to the type.

Inheritance also enables you to refine the definition by changing the default values of existing properties. Other than this you cannot change or remove properties that come form the inherited type.

Note: Icons are not inherited between models.

Inheritance Example

In the following example, the variable type ColdTemp inherits all the properties and default values from the variable type Temperature, but overrides the default value for the property UPPER.

```
VARIABLE ColdTemp USES Temperature
  UPPER: 273;
END
```

In the following example, the model HeatX inherits from the model Heater. It extends Heater by adding the variable UA and an equation to calculate Q. It also changes the default spec of Q to free, as Q is now calculated by an equation:

```
MODEL HeatX USES Heater
  UA as HTCoeff (1, Fixed);
  Q: free;
  Q=UA*DT;
END
```

Type Relatives

The inheritance relationship enables you to define a family of related types that can be substituted for each other.

For example, in a procedure definition, you list the types of the input and output variables expected in a call to that procedure. Using inheritance, you can define new variables that inherit properties from the type you have named in the procedure call argument list. You can use these new variable types in a call to the procedure, because they are recognized as descendants of the variable types you named in the procedure call list.

You can use related types to determine which ports are allowed to be connected using custom streams. You can only connect two ports if the port types are the same or are related port types. Note that if you use the built-in Connection stream, any two ports can be connected. In this special case, variables in the ports being connected are matched by name.

Type Relatives Example

The first example shows how two ports are created using inheritance. These two related ports can be connected together in the flowsheet.

```
PORT Material
  Flow AS Flow_Vol_Liq;
END

PORT NewMaterial USES Material
  Temp AS Temperature;
END

MODEL Tank1
  .
  Out1 AS OUTPUT NewMaterial;
  .
END

MODEL Tank2
  .
  In1 AS INPUT Material;
  .
END
```

```
STREAM MaterialStream;
  In1 AS Input Material;
  Out AS Output Material;
  In1.Flow = Out.Flow;
END
```

The port Material carries flow rate, and the port NewMaterial carries flow rate and temperature values. If we create instances of the two models we can connect their ports together with a stream of type MaterialStream. This is because the two ports are related by inheritance. However, the temperature value carried from the instance of Tank1 is not passed to the instance of Tank2.In the following example, the procedure TotalMassCalc is written to accept a variable of type Density as an input argument. However, the same argument can accept a variable of a type inherited from Density. In this case the variable types VaporDensity and LiquidDensity are related by inheritance to Density.

```
VARIABLE Density
  Value: 1000.0;
  Lower: 1.0e-2;
  Upper: 1.0e5;
  PhysicalQuantity: "Density";
END
VARIABLE LiquidDensity USES Density
  Value: 1000.0;
  Lower: 10.0;
  Upper: 1.0e5;
END

VARIABLE VaporDensity USES Density
  Value: 1.0;
  Lower: 1.0e-2;
  Upper: 100.0;
END

PROCEDURE TotalMassCalc
  .
  .
  INPUTS : Area, Length, Density;
  OUTPUTS: Mass;
```

```
END

MODEL Tank

  TankArea   AS Area;
  TankHeight AS Length;
  RhoLiq     AS LiquidDensity;
  RhoVap     AS VaporDensity;
  TotalMass  AS Mass;

  Phase AS PhaseParameter("Liquid");

  IF Phase == "Liquid" THEN
    CALL (TotalMass) = TotalMassCalc(TankArea, TankHeight,
RhoLiq);
  ENDIF

  IF Phase == "Vapor" THEN
    CALL (TotalMass) = TotalMassCalc(TankArea, TankHeight,
RhoVap);
  ENDIF

END
```

Component Lists

A component list contains two types of information:

- A list of component names
- A list of options associated with these components. Typically this is used to store options for calculating physical properties for mixtures of these components

A component set is a simplified version of a component list which does not include a list of physical property options.

In general you should use a full component list when using a physical properties package such as Aspen Properties, and a component set if you are not using a physical properties package.

Defining which ComponentList to use

You can define the name of the default flowsheet component list to be used for you simulation by editing the value of ComponentList in the Flowsheet LocalVariables table. By default this has the value "Default" which means the component list called Default is used.

All blocks, streams and ports have a built-in property called ComponentList. This contains the name of the component list to be used for the block, stream or port. When a new block or stream is instanced it's ComponentList property is set to the flowsheet component list name. You can change this value from the AllVariables table for the block or stream if you want to use a different component list for the block or stream.

You can use ComponentList to index arrays over the set of components to be used in the block, stream or port.

Syntax for Indexing Arrays using ComponentList

```
VariableName (ComponentList) AS VariableType;
```

VariableName ..... Name you give to a variable

VariableType..... The type of the variable

Using ComponentList Example

The following is an extract from a model that shows a mole fraction array being declared, and indexed over the components in the component list.

```
MODEL FeederPP
  :
 Z(ComponentList) AS  Molefraction;
  :
END
```

Using More Than One Component List in Models

If you are using a physical properties package such as Aspen Properties to calculate physical properties, you can use Procedure calls in your Models and Stream types to perform the property calculations. Each call automatically passes the components and options defined in the component list used by the block or stream.

This mechanism enables you to use different component lists in different blocks and streams to use different physical property options in different parts of your flowsheet.

If a model or stream needs to use more than one component list, you can define special parameters of type ComponentListName for storing these

component list names (ComponentListName is the same type that is used for the built-in ComponentList property). You can then associate the appropriate component list with each physical property procedure call by ending the call with the appropriate component list name.

When you use the model or stream you can then change the value of each component list name to that of the required component list.

Syntax for Defining Component List Names

```
ComponentListName AS COMPONENTLISTNAME;
```

ComponentListName..... The name of the parameter used to store the component list name

Using More Than One Component List in Models Example

This example shows two component lists used in the same model. These two component lists are associated with the two sides of a heat exchanger. The user of the model can then assign values to the HotSide and ColdSide component list names to reflect the fluids flowing on each side of the heat exchanger.

```
MODEL HeatExchanger
 HotSide, Coldside AS ComponentListName;
 ColdSideInlet  AS INPUT  Process (ComponentList:
ColdSide);
 ColdSideOutlet AS OUTPUT Process (ComponentList:
ColdSide);
 HotSideInlet   AS INPUT  Process (ComponentList: HotSide);
 HotSideOutlet  AS OUTPUT Process (ComponentList: HotSide);
  :
  :
 CALL (ColdSideOutlet.H) = pEnth_Mol (ColdSideOutlet.T,
   ColdSideOutlet.P, ColdSideOutlet.z) ColdSide;

 CALL (HotSideOutlet.H) = pEnth_Mol (HotSideOutlet.T,
   HotSideOutlet.P, HotSideOutlet.z) HotSide;
END
```

Built-In Type Definitions

Aspen Custom Modeler contains a number of built-in parameter and variable types.

Built-In Parameter Types

The four built-in parameter types are:

- StringParameter
- IntegerParameter
- RealParameter
- LogicalParameter

You can use these parameter types directly within type definitions, such as models. You can also inherit from them when you define your own parameter types.

The attributes of each parameter type are listed below:

StringParameter Attributes

Description	A text string that can be used to store a description of the parameter
Value	A text string that contains the value of the parameter

IntegerParameter

Description	A text string that can be used to store a description of the parameter
Value	The value of the integer parameter
Upper	The upper bound for the value (defaults to 32767)
Lower	The lower bound for the value (defaults to –32767)

RealParameter Attributes

Description	A text string that can be used to store a description of the parameter
Value	The value of the real parameter
Upper	The upper bound for the value. Default is 1.0e37.
Lower	The lower bound for the value. Default is –1.0e37.
PhysicalQuantity	The name of the physical quantity used for the parameter. Default is none.
Tag	Label for use in interfaces to online applications

LogicalParameter Attributes

Description	A text string that can be used to store a description of the parameter

Value ..... The value of the logical parameter, can be TRUE or FALSE.

Built-In Variable Type

The built-in type RealVariable is inherited by all variable types that you define. This means that all variables have the RealVariable attributes.

`RealVariable Attributes`

Attribute	Description	Default	Type
Derivative	The rate of change of the value with respect to time. This attribute is only active if the variable is differentiated in a differential equation	0.0	Real
Description	A string of text that contains summary or help information for the variable	None	Text
IsConnected	A logical property for testing whether the variable is connected to a control stream in the flowsheet. Read only	False	Boolean
IsMatched	When the variable is contained in a port, this is used to test whether connected ports have equivalent variable names. Read only.	False	Boolean
Lower	The lower bound for the variable value.	-1.0e37	Real
Physical Quantity	A string of text that contains the name of the physical quantity definition used by the variable. Read only.	None	Text
Record	Controls recording of history for a variable. This is automatically set to True when the variable is displayed on a plot or history table.	False	Boolean – True or False
Scale	The scaling factor for the variable value used by the solver	1.0	Real
Spec	A string of text that contains the specification for the variable	"Free"	Text – must be one of:- "Free", "Fixed", "Initial", "RateInitial"
Tag	Label for use in interfaces to online applications	None	Text
Units	Current UOM symbol. The available values depend on the physical quantity used for the variable		Text
Upper	The upper bound for the variable value	1.0e37	Real
Value	The value of the variable	1.0	Real

Setting Attribute Values

In the modeling language an attribute value can be set using the following syntax.

For real attributes:

e.g. Temperature.Value: 12.34;

For Boolean attributes:

e.g. Temperature.Record: True:

For text attributes:

e.g. Temperature.Spec: Fixed;

You can assign the value and spec attribute in 1 statement

e.g. Temperature: 2.25, Initial;

or only the value or the spec

e.g. Temperature: Initial;

e.g. Temperature: 2.25;

In a script the syntax is:

e.g. Temperature.Value = 12.34

For Boolean attributes:

e.g. Temperature.Record = False

For text attributes:

e.g. Temperature.Spec = "Free"

Note the full path to the variable should be included depending on the context of the script.

Getting Attribute Values

In the modeling language an attribute value can be used for example in an if statement.

e.g. if (Temperature.IsMatched) then …

e.g. if (Temperature.spec == Fixed) then …

2 Modeling Language for Simulations

This chapter describes the following:

- Using units of measurement.
- Defining tasks.
- Using task statements.
- Defining an estimation simulation.
- Defining an optimization simulation.
- Defining a homotopy simulation.

For information on defining types and models, see Chapters 3 to 4; Modeling Language for Types and Modeling Language for Models.

Using Units of Measurement

For any given physical quantity, such as temperature, you can define any number of different units of measurement (UOM), and conversions between these different units of measurement. You can group together units of measurement into UOM sets, such as SI or metric, and make these UOM sets available automatically when you start a simulation session or load a new simulation.

The units of measurement used in your models are referred to as the base units of measurement. When you write your own variable types and models, you are implicitly defining the base units of measurement. These are what are used at run time to solve your simulation. You must ensure that your model equations are written consistently in these units of measurement.

The user interface converts between the current display UOM and the base units of measurement. This means that values shown on plots and tables are automatically displayed in the current UOM and values you input on forms are automatically converted to the base units of measurement.

To determine the current units of measurement, you can either select a unit of measurement set or set the Units attribute of individual variables.

Scripts and automation also have features to support unit of measurement conversion. Tasks always work in the base units of measurement, and that all values that you enter in tasks should be in these units.

The base unit of measurement and unit of measurement conversions are defined as part of a physical quantity. A variable type is associated with a physical quantity by assigning the name of the physical quantity to the Physical Quantity attribute in the variable type definition.

The Modeler library works in Metric base units. Sets of conversions to other units of measurement are built in ready for you to use. We strongly recommend that you use this unit of measurement set for any models that you develop. You can then benefit from all of the built in unit of measurement capabilities, without having to reconfigure these.

You can edit the built in conversion sets, or add your own completely new UOM set. To perform these operations, you use automation methods from a script.

The methods you can use are:

Automation Method	Description
AddUomSet	Adds a unit of measurement set.
AddPhysicalQuantity	Adds a physical quantity to the system and specifies its base units of measurement.
DefineConversion	Defines a conversion between base units and another unit of measurement.
DefaultDisplayUOM	Defines the units of measurement in which a physical quantity is displayed, for an existing units of measurement set.
SelectUomSet	Enables you to select an existing unit of measure set.
ConvertToBaseUom	Converts a value from its current display unit of measurement to its base unit of measurement.
ConvertFromBaseUom	Converts a value from its base unit of measure to its current display unit of measure.

Specifying a Physical Quantity for Variables

You specify the physical quantity to be used as part of a variable type or parameter type definition. The physical quantity:

- Defines the base unit of measure used for all variables of that type
- The unit of measure conversions available for all variables of that type

Syntax for Defining a Physical Quantity

The syntax for defining a physical quantity for a variable type or parameter type is:

```
PHYSICALQUANTITY: "PhysQuanName";
```

PhysQuanName ..... Name of a physical quantity

Physical Quantity Examples
The following example shows how the physical quantity energy is assigned to the variable type holdup_heat:

```
VARIABLE holdup_heat
  value: 1;
  lower: -1E5;
  upper: 1E5;
  PhysicalQuantity: "Energy";
END
```

Defining Physical Quantities, Units of Measurement, and Conversions

You can define new physical quantities, unit of measurement conversions and unit of measurement sets. You do this by writing a Visual Basic script.

Defining the UOM Object

Start your script by defining a reference to the UOM object.

Syntax for Defining UOM Object

```
DIM UOM
SET UOM = ActiveDocument.uom
```

UOM	A Visual Basic variable. You can choose any valid name for this variable. The following examples assume you have chosen the name UOM.

Defining Physical Quantities

To use units of measurement, you need to define the physical quantities used in your simulation.

Syntax for Defining Physical Quantities

The syntax for defining a physical quantity is:

```
bOK = UOM.ADDPHYSICALQUANTITY("PhysicalQuantityName",
"BaseUnitSymbol", AspenPlusQuantity, AspenPlusUnit)
```

bOK	Dummy boolean variable to accept the return code from the function call
UOM	Previously defined reference to the UOM object
PhysicalQuantityName	The name of the physical quantity
BaseUnitSymbol	The symbol for the base units of

	measurement for this physical quantity
AspenPlusQuantity.....	The integer number of the corresponding physical quantity in Aspen Plus®, as entered in the Aspen Plus file rcunits.dat.
AspenPlusUnit.....	For the given Aspen Plus physical quantity, this is the integer number of the Aspen Plus unit that matches the base units for the physical quantity.

Note: AspenPlusQuantity and AspenPlusUnit are optional and are only used if you export models from Aspen Custom Modeler for use in Aspen Plus.

You can use the full path for the Uom object instead of defining the UOM variable. For example:

```
bOK = ActiveDocument.Uom.AddPhysicalQuantity("Length", "m")
```

If you define UOM once, you can save typing the full path each time.

Physical Quantity Example

The following example creates physical quantities for length and temperature with base units of measurement of **meters** and **degrees Celsius** respectively.

```
bOK = UOM.AddPhysicalQuantity("Length", "m")
bOK = UOM.AddPhysicalQuantity("Temperature", "C")
```

Defining Conversions between the Units of Measurement

After defining the physical quantities and base units used in your simulation, you can define additional units of measurement. A unit of measure consists of a symbol for the unit, and the factors for conversion to base units.

Syntax for Defining Conversions between Units of Measurement

The syntax for defining the conversion is:

```
bOK = UOM.DEFINECONVERSION("PhysicalQuantityName",
"UomSymbol", Multiplier, Offset)
```

bOK.....	Dummy boolean variable to accept the return code from the function call
UOM.....	Previously defined reference to the UOM object
PhyscialQuantityName.....	Name of a previously defined physical quantity
UomSymbol.....	The symbol of the UOM to which you are making the conversion

	from the base UOM
Multiplier	The factor by which you need to multiply the UOM value to convert to the base UOM value
Offset	The difference you need to make to the UOM value to convert to the base UOM value

The conversion is made using the Multiplier and Offset in the following equation:

*BaseUnit = Multiplier * UomSymbolUnit + Offset*

Units of Measurement Conversion Example

The following example defines conversions for the physical quantities Temperature and Length. The base units of measurement defined in the physical quantity definitions are "C" for temperature, and "m" for length.

```
bOK = UOM.DefineConversion("Temperature", "K", 1.0, -
273.15)
bOK = UOM.DefineConversion("Temperature", "F", 0.5556, -
17.7778)
bOK = UOM.DefineConversion("Length", "ft", 0.3048, 0.0)
```

Creating Unit of Measurement Sets

Unit of measurement sets enable you to group together related units of measurement. The automation method AddUomSet enables you to add a new UOM set name to your simulation. You can create a new UOM set using the following syntax in a Visual Basic script:

```
bOK = UOM.ADDUOMSET("UomSetName")
```

bOK	Dummy boolean variable to accept the return code from the function call
UOM	Previously defined reference to the UOM object
UomSetName	The name you give to the UOM set

Creating Units of Measurement Sets Example

The following example creates three different units of measurement sets: for SI units, metric units, and English units.

```
bOK = UOM.AddUomSet("SI")
bOK = UOM.AddUomSet("Metric")
bOK = UOM.AddUomSet("Eng")
```

Defining the Displayed Units of Measurement for the Units of Measurement Set

After you have created a unit of measurement set, you can choose the unit of measurement to be used for each physical quantity within the unit of measurement set. These are the units of measurement that quantities will be displayed when the units of measurement set is selected.

Syntax for Defining the Displayed Units of Measurement

The syntax for defining the displayed units is:

```
bOK = UOM.DEFAULTDISPLAYUOM( "UOMSet",
"PhysicalQuantityName", "UomSymbol")
```

bOK	Dummy boolean variable to accept the return code from the function call
UOM	Previously defined reference to the UOM object
UomSet	The name of a previously defined UOM set
PhysicalQuantityName	The name of a previously defined physical quantity
UomSymbol	The UOM to be used for the physical quantity within the UOM set

Note: If a default display unit of measure is not specified for a physical quantity within a UOM set, then that physical quantity will be displayed in the base units when using that UOM set.

Defining the Displayed Units of Measurement Example

The following example defines the displayed units for length in the Eng UOM set as ft:

```
bOK = UOM.DefaultDisplayUOM("Eng","Length","ft")
```

Selecting a Unit of Measurement Set for the Simulation

- Once you have defined a UOM set, you can select it from the Units of Measurement item on the Tools menu. You can also select it from a Visual Basic script using the syntax:

```
ActiveDocument.Uom.SelectUomSet("UomSetName")
```

Selecting a Units of Measurement Set Example

The following example shows how to switch to the Eng units of measurement set, using a Visual Basic script:

```
ActiveDocument.Uom.SelectUomSet("Eng")
```

Running the Units of Measurement Script Automatically

You can ensure that the Visual Basic® units of measurement script runs automatically each time you start a simulation session or load a new simulation. This means you can write a script that defines all the units of measurement sets and conversions you need and ensure that the conversions are available each time you run a simulation.

To create a UOM script that runs automatically, you need to save the script text to a file called:

OnNewDocumentScript.vb

You must place this file in the bin directory of your installation. For example:

```
C:\Program Files\AspenTech\Aspen Custom Modeler 11.1\bin
```

A default version of this file already exists, and defines the standard unit of measurement sets available in Aspen Custom Modeler. You can choose to replace these, but we would recommend that instead you append your changes to the end of this file.

In addition, the Open Document event will run a script called OnLoadSimulation located under the Flowsheet in the simulation being loaded. This allows you to make changes that are specific to the simulation being loaded e.g. simulation specific units of measurements and screen layouts.

Converting Between Units of Measurement

You can convert values in Microsoft Visual Basic scripts between your base units of measurement and your current units of measurement.

Syntax for Converting Between Units of Measurement

Use the following syntax in your Visual Basic Scripts:

```
ActiveDocument.Uom.ConvertToBaseUom ("BaseUom",
CurrentUomValue)
```

and

```
ActiveDocument.Uom.ConvertFromBaseUom ("BaseUom",
BaseUomValue)
```

BaseUom.....	The base unit of measurement name
CurrentUomValue.....	The value of the variable in the currently selected unit of measurement
BaseUomValue.....	The value of the variable in the base unit of measure

Converting Between Units of Measurement Example

In this Microsoft Visual Basic script example, the current UOM set is English units and the base UOM set is metric. A value of temperature of 60 Celsius is converted to the current temperature unit of measure, Fahrenheit. A value of

length of 5280 feet is converted to the base unit of measure, meters.

```
Dim TempInF, LengthInMeters, UOM
Set UOM = Application.Simulation.Uom
TempInF = UOM.ConvertFromBaseUom("C", 60.0)
LengthInMetres = UOM.ConvertToBaseUom("m", 5280.0)
```

Using the Units of Measurement of a Control Variable

Normally variables acquire units of measurement from the variable type that they use. However it is sometimes useful to have a variable acquire it's units of measurement from another variable in the same model or flowsheet constraints section.

Syntax for Using the Units of Measurement of another Variable

```
MODEL
  VariableName1 AS VariableType1;
  VariableName2 AS VariableType2 (UseUOMOf:
"VariableName1");

END
```

Example of Using the Units of Measurement of a Control Variable

This example shows how other variables in a controller model can take the units of measurement of a variable in a different block that is connected to a control variable.

```
MODEL PI_Controller
 .
 PV AS INPUT Control_Signal;
 PVMin AS Control_Signal (Fixed, UseUOMOf: "PV");
 PVMax AS Control_Signal (Fixed, UseUOMOf: "PV");
 .
END
```

In this example, if a variable of type Temperature is connected to the input control variable PV using the ControlSignal stream type, the variables PV, PVMin and PVMax are all displayed in the same units of measurement as the connected variable.

Advanced Use of the UseUOMOf Property

The UseUOMOf property is used to declare that one variable uses the same units of measurement of another variable. For introductory information on the UseUOMOf property, see Using the Units of Measurement of a Control Variable

on page 2-42. The UseUOMOf property is also capable of more complex associations.

See Also

Using Variables whose UOM is Produced from the UOMs of Other Variables

Applying only Scaling to a Variable

Using Variables whose UOM is Determined from the UOMs of Other Variables

You can use the UseUOMOf property for a variable whose UOM is a product or division of the UOMs of other variables.

Example of Using Variables whose UOM is Determined from the UOMs of Other Variables

The following example is a model that calculates the ratio and product of two inputs:

```
Model RatioBlock
  input1 as input RealVariable(fixed);
  input2 as input RealVariable(fixed);

  ratio as output RealVariable(UseUOMOf: "input1/input2");
  prod as output RealVariable(UseUOMOf: "input1*input2");

  ratio*input2 = input1;
  prod = input1*input2;
End
```

Notes:

- The 'ratio' variable is declared as using the UOM of input1 divided by the UOM of input2. If, for example, input1 is connected to a temperature variable displaying in Fahrenheit and input2 to a length variable displaying in feet, the display units of measurement will be Fahrenheit/ft. Note that in this case the conversion for display units to base units (Centigrade/meter) will not include the offset between Fahrenheit and Centigrade: only the scaling is applied.
- The 'prod' variable is handled in a similar way to the ratio variable except in this case a "*" is used to reflect the multiplication operation.

Applying only UOM Scaling to a Variable

You can use UseUOMOf to apply only the unit of measurement scale factor to a variable.

Example of Applying only UOM Scaling to a Variable

The following example is a model that calculates the difference between two variables:

```
Model RatioBlock
  input1 as input RealVariable(fixed);
  input2 as input RealVariable(fixed);

  diff as output RealVariable(UseUOMOf: "ScaleOf: input1");

  diff = input1 - input2;
End
```

In this example, if the 'diff' variable was simply declared to use the UOM of input1 then when converting to display units, the conversion offset would be inappropriately applied (for example, a zero Centigrade temperature difference would be converted to 32.0 in Fahrenheit). To ensure that only the scale is applied, this example uses the "ScaleOf:" qualifier before the name of the variable.

Accessing Variable Values in Alternative UOMSets

You can access the values of variables in alternative UOM Sets from the current UOMSet through automation.

This means you can access variable values in a Microsoft® Visual Basic® Script in a different unit of measure and use the alternative unit of measure for intermediate calculations or results output.

Syntax

The syntax for a script at Flowsheet level is:

```
BlockName.VariableName.VALUE("UOMName")
```

The syntax for the script at Model level is:

```
VariableName.VALUE("UOMName")
```

BlockName	Name of a block on your flowsheet
VariableName	Name of a variable in the block
UOMName	Can take one of the following values:
BaseUnits	Returns the variable value in base units

CurrentUnits	Returns the value in the current units of measurement
UnitName	The name of the unit of measurement in which the value is to be returned

Example of Accessing Variable Values in Alternative UOM Sets

For a Visual Basic Script at Flowsheet level:

```
Application.Msg "Temperature in base units is " +
B1.Out1.Temp("BaseUnits")
Application.Msg "Temperature in current units is " +
B1.Out1.Temp("CurrentUnits")
Application.Msg "Temperature in Celsius is " +
B1.Out1.Temp("C")
Application.Msg "Temperature in Fahrenheit is " +
B1.Out1.Temp("F")
```

If the BaseUOMSet is Metric and the CurrentUOMSet is US units, the output from the script is:

```
Temperature in base units is 22.2222
Temperature in current units is 72
Temperature in Fahrenheit is 72
Temperature in Celsius is 22.2222
```

Defining Tasks

A task is a set of instructions that defines a sequence of actions that take place during a dynamic simulation. For example, you can use tasks to:

- Define disturbances in feed conditions or controller set points at a pre-defined time
- Trigger events when certain conditions are met. For example, you can use a task to provide basic level control in a tank by closing an output valve when the fluid level falls below a certain value.

You can define two kinds of tasks:

Task Type	Started By	Explorer Icon
Event-driven	A start condition, which is written as a conditional expression.	
Callable	Being called from other tasks, either an event-driven task or another callable task. Does not have a start condition.	

There are three locations where you can define a task:

Location	Can be Event-Driven	Can be Callable
Model	Yes	Yes
Flowsheet	Yes	Yes
Library Task folder	No	Yes

Event-Driven Tasks

Event-driven tasks are associated with a defined event and execute when that event occurs.The events with which a task can be associated are:

Event	Example
A specified simulation time	AT 20.0
A logical expression involving variable values	tank.level >= 2.5

Variables can appear on both sides of the conditional expression.

The allowable comparison operators are >=, >, = =, <=, <>.

= = means equal to, and <> means not equal to.

For time expressions you can use only >=, > or = =.

Event-Driven Task Syntax

The syntax for an event-driven task is either:

```
TASK TaskName RUNS AT Time
  TaskStatements
END
```

or

```
TASK TaskName RUNS ONCE WHEN Condition
  TaskStatements
END
```

TaskName ..... Name of the task

Condition ..... A logical expression testing the value of simulation variables (including time)

Time An expression evaluating to a positive real value representing a simulation time

ONCE Optional keyword (see following Remarks)

TaskStatements ..... The sequence of task statements that are executed when the task is triggered

You can also use the following system generated events, in place of WHEN *Condition*, to trigger your tasks:-

- BEFORE INITIALIZATION
- AFTER INITIALIZATION
- BEFORE STEP
- AFTER STEP
- AFTER ERROR

The associated task triggers when the system generated event occurs. That is; before or after simulation initialization, before or after a dynamic run takes a step, or when any error occurs.

Note: An event-driven task cannot take input parameters.

Event-Driven Tasks Remarks

If you include the ONCE keyword in the task start condition, the task runs only once during the simulation, when then start condition first becomes true. This can be used to model irreversible events.

A task without the ONCE keyword starts whenever the start condition becomes true.

The ONCE keyword is implied by AT Time.

The trigger RUNS ONCE When Time == 20.0 is the same as RUNS AT 20.0.

Note: You can also use a SWITCH statement in a model definition to model an irreversible process.

Examples of Event-Driven Tasks

In the first example, at time 5.0, the flow to a vessel is set to zero. The tank empties until the fluid level is below 0.01, when the inlet flow is resumed.

```
TASK test1 RUNS AT TIME 5.0
  Input1.Flow: 0.0;
  WAIT FOR Level <= 0.01;
  Input1.Flow: 5.0;
END
```

In the second example, the inlet flow is set to 3.0 whenever the level is below 0.01. This can happen at any time in the simulation.

```
TASK test2B RUNS WHEN Level <= 0.01
  Input1.Flow: 3.0;
END
```

In the following example, the event-driven task is defined to run once only. In this case a bursting disk is modeled so that once the burst pressure is exceeded, the disk remains open:

```
TASK Burst RUNS ONCE WHEN Tank.Press > 9.0
  Valve.CV: 1.0E6;
END
```

Finally, in this example, the associated task triggers after an initialization occurs.

```
Task mytask RUNS AFTER INITIALIZATION
  Input.Flow: 5.0;
END
```

Callable Tasks

Callable tasks are called by other tasks instead of being triggered by an event. You can optionally define callable tasks with parameter call lists, with the calling task passing the parameter values.

You can make calls to tasks within models, to tasks in a tasks folder, or other tasks in your flowsheet. You cannot call a task in a model from a flowsheet task, and you cannot call tasks in the flowsheet from a task in a model or task folder.

Callable Task Syntax

The syntax for a callable task is:

```
TASK TaskName (ParameterList)
  TaskStatements
END
```

TaskName	The name of the task
ParameterList	An optional list of parameters to be passed to this task when it is called
TaskStatements	The task statements that are executed when the task is run

Use a CALL statement to call a callable task.

CALL Statement Syntax

The syntax for the CALL statement is:

```
CALL TaskName (ParameterList);
```

TaskName	The name of the task being run
ParameterList	A list of assignments to the parameters of the called task. Only

required if the called task has on or more input parameters.

Example of a Callable Task

The following example shows two callable tasks being called from an event-driven task:

```
TASK SubTaskA
  Input1.Flow: 4.0;
END


TASK SubTaskB
  Input1.Flow: 1.0;
END


TASK Test RUNS AT Time 1.0
  Input1.Flow: 5.0;
WAIT FOR Level >= 3.0;
  CALL SubTaskB;
WAIT FOR Level <= 0.4;
  CALL SubTaskA;
END
```

Callable Task Parameter Lists

You can pass parameters from a task to a callable task. Use this to pass variable names or calculated values from one task to another. You can use parameterized tasks to create generic tasks that are not dependent on a particular block name, and then pass information that is specific to a block to that generic task.

Parameterized Task Syntax

The ParameterList has the following syntax:

```
(ArgumentName1 AS ArgumentType1; ArgumentName2 AS
ArgumentType2; ...)
```

ArgumentName.....	The name of the variable or parameter within the callable task
ArgumentType	The type of the argument. This can be RealVariable, RealParameter, IntegerParameter or StringParameter.

Note: Use argument type RealVariable to pass in a simulation variable. Use RealParameter, IntegerParameter and StringParameter to pass in parameters defined either in the simulation flowsheet, or in the calling task.

Callable Task Parameters Example

The example shows how you can write a callable task that uses parameters passed from the task that calls it. The information passed to the callable task is the name of a variable to be ramped, the target value of the ramp, the duration of the ramp and the type of ramp used. The task GenericRamp can be called from any model task or from a flowsheet task.

```
TASK GenericRamp (VarName AS RealVariable; Target AS
RealParameter; Period AS RealParameter; RampType AS
StringParameter)
 IF RampType == "Linear" THEN
  RAMP (VarName, Target, Period);
 ENDIF
 IF RampType == "Scurve" THEN
  SRAMP (VarName, Target, Period);
 ENDIF
END


TASK DoTheRamp RUNS WHEN B4.Level >= 10.0
 TargetValue  AS RealParameter(0.0);
 RampDuration AS RealParameter(0.1667);
 RampType     AS StringParameter("Scurve");
 CALL GenericRamp(B1.Inlet.Flow, TargetValue, RampDuration,
RampType);
 WAIT FOR B4.Level <= 1.0;
 TargetValue:  4.25;
 RampDuration: 0.2;
 RampType:    "Linear";
 CALL GenericRamp(B1.Inlet.Flow, TargetValue, RampDuration,
RampType);
END
```

Tasks and Opening Simulations

The state of any active tasks is not stored when you save a simulation, and so cannot be restored when you load the simulation. Therefore it is recommended that you do not save your simulation at a time when any tasks are currently running, or that if you do you ensure that after loading you run the simulation from time=0.

Using Task Statements

Task statements can perform the following tasks:

- Assign a value to a variable.
- Define a conditional expression that depends upon the value of a simulation variable.
- Ramp the value of a simulation variable.
- Suspend or restart the task.
- Create snapshots.
- Execute parallel tasks.
- Print.
- Suspend a simulation.
- Invoke a script from the task.
- FOR loops in tasks.
- Task Language Quick Reference

Assigning a Value to a Variable in Tasks

You can define parameters and sets that are local to a particular task. The following types can be used:

- IntegerParameter
- RealParameter
- LogicalParameter
- StringParameter
- IntegerSet
- StringSet

The syntax is the same as for defining parameters and sets in models.

You can assign values to local parameters, model parameters, and model variables using the **:** (colon) assignment operator.

Example of Assigning a Value to a Variable

The following example shows how a value is assigned to a variable:

```
TASK Test3 RUNS WHEN h < 0.001
```

```
 Fin as RealParameter;
 Fin: 21.5;
 Tank1.flowin: Fin;
 Tank1.tempin: 50.0;
END
```

UOM Conversions in Tasks

Tasks are usually written in the base units. However, limited UOM conversions are possible with the following syntax:

VarToAssign: Expression {UOMString};

VarToAssign.....	The task parameter or model variable to be assigned, which has Units of measurement defined.
Expression.....	A Real Expression specifying the value to assign.
UOMString.....	A string specifying the Units of measurement of *Expression*. This must be a valid unit of measurement for *VarToAssign*.

Example of Assigning a Value to a Variable with Units of Measurement

```
B1.Temperature: 25 {C};
```

Using the Random Function

In tasks, you can use the RANDOM function in an assignment, with either a UNIFORM or NORMAL distribution.

The syntax is as follows:

VarToAssign : RANDOM(MeanValue, RangeValue, Distribution);

VarToAssign.....	The task parameter or model variable to be assigned.
Mean Value.....	A Real Expression specifying the mid-point of the possible values to assign.
RangeValue.....	A Real Expression specifying the range of the values to assign.
Distribution.....	UNIFORM or NORMAL

Note

- If the Distribution is NORMAL, the RangeValue is taken to be the standard deviation of a Gaussian Distribution about the specified MeanValue.
- If the Distribution is UNIFORM, the returned value will be in the range MeanValue-RangeValue to MeanValue+RangeValue.

Example of Assigning a Value to a Variable with Units of Measurement

```
S1.Signal: RANDOM(s1.Signal,0.5,NORMAL);
```

The RANDOM function may be used in an expression:

```
HeatIn: Q + RANDOM(10,2.0,UNIFORM);
```

The IF Construct

Use the IF construct to control which statements are executed dependant upon whether a conditional expression is true or false.

IF Construct Syntax

The IF construct has the following syntax:

```
IF Condition THEN
  TaskStatements1
ELSE
  TaskStatements2
ENDIF
```

Condition	A conditional expression
TaskStatements	The statements to be executed if the condition is true.
TaskStatements2	The statements to be executed if the condition is false.

IF Construct Remarks

The ELSE and TaskStatements2 are optional.

Example of Defining a Conditional Equation

The following example assigns a value to one variable dependant upon the value of another variable:

```
TASK OutFlow RUNS WHEN time > 0.001
 IF Tank1.Level > 0.1 THEN
  Tank1.Flowout: 1.0;
```

```
 ELSE
  Tank1.Flowout: 0;
 ENDIF
END
```

Ramping the Value of a Simulation Variable in Tasks

There are two ramp functions that you can use to change the value of a variable over a period of time.

Ramp Function Syntax

```
RAMP (Variable, Target, Period, Ramptype);
SRAMP (Variable, Target, Period, Ramptype);
```

Variable	The name of the variable whose value is to be ramped. The variable must have a specification of fixed.
Target	The value to which the variable's value is to be changed
Period	The period of time over which the ramp takes place
Ramptype	Can be either DISCRETE or CONTINUOUS. CONTINUOUS ramps are re-calculated whenever the integrator moves its time. Continuous ramps incur a small performance penalty, but can be considerably more accurate when used with a variable step integrator. DISCRETE ramps are re-calculated at communication points only. The ramptype may be omitted; in which case the default ramptype of CONTINUOUS is used.

RAMP is a simple linear ramp function . SRAMP is a sinusoidal ramp, which gives a smoother S-shaped curve for the ramped variable. With SRAMP, if the ramp is decreasing the value of a variable then the ramp follows the shape of a sinusoidal curve between 0 and pi. If the target is an increase then the curve follows the shape of a sinusoidal curve between pi and 2 x pi.

Example of Ramping the Value of a Variable

The following example shows the use of ramping:

```
TASK Task4 RUNS AT TIME 4.0
 RAMP (Input1.flow, 5.0, 2.0);
```

```
 /* Flow changes to 5.0 linearly over a period of  2 time
units */

 SRAMP (Input1.temp, 15.0, 3.0);

 /* Temperature changes with an S-shaped curve to 15.0 over
a period of 3 time units */

END
```

Suspending and Restarting a Task

You can suspend the execution of a task until a particular condition is met. You can use the following statements to suspend a task:

Statement Syntax	Use
WAIT RealExpression;	Suspends the execution of the task for the number of time units represented by RealExpression.
WAIT FOR Condition;	Suspends the execution of the task until Condition is met.
RESTART AFTER RealExpression;	Suspends the execution of a task for the number of time units represented by RealExpression, and then ends the task. The task will then restart as soon as its start condition is true.
RESTART WHEN Condition;	Suspends the execution of a task. When Condition is met, the task ends. The task will then restart as soon as its start condition is true.
RESTART;	Ends the execution of a task. The task will then restart. Execution will continue from the first statement after the start condition.
RETURN;	Suspends the execution of a task. The task will then restart as soon as its start condition is true.

Note:

- You can use RESTART AFTER, RESTART WHEN, and RESTART in event-driven tasks only. You cannot use these statements in callable tasks.
- No statements after RESTART will be executed, therefore these statements should normally be the last statement in a task.

Examples of Suspending and Restarting Tasks

The first example shows how to use a WAIT statement to suspend a task:

```
TASK TurnOffFeeds RUNS AT TIME 10
 valve1.position: 0.0;
 WAIT 0.2;
 valve2.position: 0.0;
END
```

The following example shows the use of a WAIT FOR statement that causes the task to pause until a condition becomes true.

The task starts when a fluid volume in a vessel falls below a certain value. The feed flow rate to the task is immediately increased to 10.0 until the fluid volume in the tank reaches a certain value, at which point the feed flow rate is returned to its original value:

```
TASK WaitForTest RUNS WHEN Tank1.Vol =< 0.5
 OriginalFlow AS RealParameter;
 OriginalFlow: Tank1.Input1.Flow;
 Tank1.Input1.Flow: 10.0;
 WAIT FOR Tank1.Vol >= 1.0;
 Tank1.Input1.Flow: OriginalFlow
END
```

The following example shows the use of a RESTART WHEN statement.

The task starts running at a time=10. The feed flow rate ramps up and the feed component molefractions are changed. Once a certain level of ethanol is reached in the tank, the feed compositions are changed and the flow rate ramps down. The task now restarts, based on the level of ethanol in the tank. The cycle carries on repeating.

```
TASK RestartWhenTest RUNS AT TIME 10.0
 RAMP (Tank1.Input1.Flow, 10.0, 5.0);
 Tank1.Input1.x("Water")  : 0.1;
 Tank1.Input1.x("Glucose"): 0.9;
 WAIT FOR Tank1.Output1.x("Ethanol") >= 0.6;
 Tank1.Input1.x("Water")  : 1.0;
 Tank1.Input1.x("Glucose"): 0.0;
 RAMP (Tank1.Input1.Flow, 1.0, 3.0);
 RESTART WHEN Tank1.Output1.x("Ethanol") <= 0.05;
END
```

Creating Snapshots in Tasks

You can use a task to create a snapshot of the current values of the variables in the dynamic simulation.

You can use this to record the simulation values when a certain process condition is reached.

CREATE SNAPSHOT Syntax

The syntax for creating a snapshot is:

```
CREATE SNAPSHOT SnapshotName;
```

SnapshotName..... The name of the snapshot to be created

Creating Snapshots Example

The following example shows how to create a snapshot at the moment a process condition is reached. You can now rewind or apply the values of all the variables in the simulation at the point when the process condition is achieved.

```
TASK Task2 RUNS AT Time 10.0
 Flowin.flow: 5.0;
 WAIT FOR Level >= 2.5;
 CREATE SNAPSHOT "Tank full";
END
```

Executing Tasks in Parallel

You can execute a number of callable tasks in parallel using the PARALLEL construct. Each action is executed until it completes. The parallel section completes when all of the Calls are complete.

PARALLEL Syntax

The syntax for grouping actions in parallel is:

```
PARALLEL
  CALL Task1;
  CALL Task2;
   :
  CALL Taskn;
ENDPARALLEL;
```

Task..... The name of a callable task to be called in parallel

Example of Executing Tasks in Parallel

The following example shows how to use of PARALLEL to call three sub-tasks simultaneously:

```
TASK P1
 RAMP(Input1.Flow, 2.5, 4.0);
END

TASK P2
 RAMP(Input2.Flow, 0.5, 5.0);
END

TASK P3
 RAMP(Input3.Flow, 5.5, 3.0);
END

TASK Task1 RUNS AT TIME 1.25
 PARALLEL
  CALL P1;
  CALL P2;
  CALL P3;
 ENDPARALLEL;
 Input4.Flow: 0.0;
END
```

All three Ramps start at time 1.25. The last task to finish is Task P2, which completes at 6.25, at which time Input4.Flow is changed to 0.0.

Note: If the event task Task1 is created at flowsheet level, the 3 callable tasks P1, P2 and P3 should be created as separate tasks at flowsheet level. You cannot put the definitions of P1, P2 and P3 in with the definition of Task1.
However if these tasks were owned by a model definition, all the task definitions shown above could be included in the model definition. Any block that used that model definition would have its own copy of the tasks. The event task could then be activated separately for each block.

PRINT Statements in Tasks

You can print a message from a task to the Simulation Messages window.

PRINT Syntax

```
PRINT text1 + text2 + … ;
```

text	A quoted text string, or a StringParameter

PRINT Statement Examples

The following example shows the use of the print statement within a task:

```
TASK RampFlow RUNS AT Time 1.0
 PRINT "Start Task RampFlow";
 RAMP (Input1.Flow, 2.5, 5.0);
 PRINT "Task RampFlow Finished";
END
```

The following example shows how to use CONVERTTO to convert to a string:

```
TASK MyTask
 s2 as StringParameter;
 s2: CONVERTTO(string, Input1.Flow);
 print "Value of Input1.Flow is " + s2;
END
```

Note that CONVERTTO cannot be used directly within a PRINT statement.

Pausing a Simulation from a Task

You can use a PAUSE statements to suspend a simulation.

PAUSE Syntax

The syntax for suspending a simulation is:

```
PAUSE;
```

When a pause statement executes, the simulation is placed in a paused state, but remains active, and can trigger again once the run is continued. If the simulation is then run on from the paused time, the task continues execution at the next statement.

Example of Pausing a Simulation

The following example makes a step change to a variable, waits for a process condition to be reached and then pauses the simulation:

```
TASK MyTest1 RUNS AT TIME 5.0
 FlowIn.Flow: 10.0;
```

```
 WAIT FOR FluidHeight >= 4.5;
 PAUSE;
END
```

Invoking a Script from a Task

You can use the Invoke command within a task to invoke a script.

INVOKE Syntax

```
INVOKE (Output1, Output2, ...) : ScriptLocation.ScriptName
(Input1, Input2, ...);
```

ScriptLocation	The optional location of the script to be run. If omitted the script is assumed to be in the same location as the Task. *ScriptLocation* can be one of: **LIBRARY** - The script is in the Custom Modeling, Scripts folder. ***LibraryName*** - The script is in the Scripts folder of the named library. The library must be open when the task runs.
ScriptName	The name of the script to be run.
Input.....	An input argument to the script. Can be a integer, real or string value, or the name of a variable or parameter. Use of input arguments is optional.
Output	An output argument from the script. This is the name of a variable or parameter. Use of output arguments is optional.

Input parameters passed to the script will be accessible in the script as members of the Inputs collection. For example the first input argument will be accessible in the script as Inputs(1).

Output parameters from the script will be accessible in the script as members of the Outputs collection. For example the first input argument will be accessible in the script as Outputs(1). On entry to the script the values of the Outputs collection are initialized to the current values of the output variables/parameters.

The task will continue with the next statement once the script has completed

Examples of Invoking a Script from a Task

The following example invokes a script called ShowForm. The script has no arguments:

```
INVOKE ShowForm;
```

The following example calls the CalculateRamp script to determine how far to ramp a variable:

```
TASK controlstep RUNS AT Time 1.0
 target   as realparameter;
 interval as realparameter;
 INVOKE (target, interval):
CalculateRamp(tank1.flowin.flow);
 RAMP(tank1.flowin.flow, target, interval);
END
```

The CalculateRamp script could take the form:

```
outputs(1) = inputs(1) * 2
outputs(2) = 3.0
```

FOR Loops in Tasks

You can use a FOR loop in a task to execute a loop a specified number of times.

FOR Loop syntax

```
FOR Index IN SetExpression DO
  TaskStatements;
ENDFOR;
```

Index	Loop Index, which will be an integer or string, depending on the type of the set expression.
SetExpression	A set of values for the Loop Index to take.
TaskStatements	The statements to be executed.

FOR Loop Remarks

When the SetExpression is an integer set, Index starts with the lowest value in the set on the first iteration, and takes successively greater values on each subsequent iteration.

When the SetExpression is a string set, Index takes a different element from the set on each iteration, but the order in which the elements are taken is undefined.

FOR Loop example

The following example uses a task to follow a time profile in a parameter array. The array DataArray is assumed to be indexed by the integer set DataPoints, both of which are declared in the enclosing model.

```
Task FollowData Runs Once Always
  For iPoint in DataPoints Do
    RAMP(X,DataArray(iPoint),1.0);
  EndFor;
End // Task FollowData
```

REPEAT Loops in Tasks

You can use a REPEAT loop in a task to execute a loop until a condition is met. The statements in the body of the loop will always execute at least once.

REPEAT Loop syntax

```
REPEAT
   TaskStatements;
UNTIL Conditon;
```

REPEAT Loop example

The following example adds a random value to the level, then waits for a time of 1. The loop continues executing until the volume exceeds 10. You could imagine that this is modeling someone filling a tank using a bucket – but it might be better if the wait period were also a random value!

```
Task RepeatExample Runs At 0.0
  Repeat
    Volume:Volume + Random(1,0.3,NORMAL);
    Wait 1;
  Until volume>10;
End // Task RepeatExample
```

WHILE Loops in Tasks

You can use a WHILE loop in a task to execute a loop while a condition is met. If the condition is initially FALSE, the statements in the body of the loop are not executed.

WHILE Loop syntax

```
WHILE Expression DO
   TaskStatements;
ENDWHILE;
```

WHILE Loop example

The following example increments the volume of one component in a mixture while the mixture is acidic. How many times the loop executes depends on the initial PH of the mixture, and the ratio of the volumes of the components. The loop includes a wait to allow time for the components to react.

```
TASK WhileExample RUNS AT 0.0
while (PH < 7.0) do
  vol("A") : vol("A") + 0.05;
  wait 0.1;
endwhile;
End // Task WhileExample
```

Task Language Quick Reference

The following is a quick reference guide to the task language.

```
PARAMETER ParameterType USES BuiltInType
  VALUE: DefaultValue;
  VALID AS SetType (ValidList);
END
```

Defines a parameter type.

```
SetName AS SetType (SetContents)  ;
Defines a set
```

```
Variable.Attribute : value;
```

Assigns a value to variable or parameter.

```
IF Condition THEN
  TaskStatements1
ELSE
  TaskStatements2
ENDIF
```

Controls task statements based on a condition.

```
CREATE SNAPSHOT SnapshotName;
```

Creates a snapshot

```
PARALLEL
  CALL Task1;
  CALL Task2;
   :
ENDPARALLEL;
```

Runs tasks in parallel.

```
PAUSE;
```

Pauses the task and the simulation. The task resumes if the simulation is continued.

```
PRINT text1 + text2 + … ;
```

Outputs message.

```
RAMP (Variable, Target, Period, Ramptype);
SRAMP (Variable, Target, Period, Ramptype);
```

Ramps a variable value.

```
WAIT RealExpression;
```

Suspends the execution of the task for the number of time units.

```
WAIT FOR Condition;
```

Suspends the execution of the task until Condition is met.

```
RESTART AFTER RealExpression;
```

Suspends the execution of a task for the number of time units.

```
RESTART WHEN Condition;
```

Suspends the execution of a task.

```
CALL TaskName (ParameterList);
```

Calls another task.

```
TASK TaskName RUNS ONCE WHEN Condition
TASK TaskName RUNS WHEN Condition
TASK TaskName RUNS AT time
```

Runs a task based on a condition.

```
INVOKE (Output1, Output2, ...) : ScriptLocation.ScriptName
(Input1, Input2, ...);
```

Invokes a script.

Defining Flowsheet Constraints

You can define the following in a flowsheet constraint definition:

- Variables
- Parameters
- Equations and procedure calls
- Assignments
- Inequalities

You can use the flowsheet constraints section to write any equations that include variables from different blocks on your flowsheet.

To enter flowsheet constraints, within Simulation Explorer go to the Flowsheet item within Flowsheet and select Edit.

The syntax used for declaring variables, parameters, equations and procedure calls and assignments is the same as that used within model definitions, as described in chapter 4. The syntax for in equalities is described in Defining Optimization Constraints.

To refer to variables within blocks in the flowsheet you need to include the block name. For example to refer to the variable Tout in the block HX1 use:

```
HX1.Tout
```

You can also refer to variables within streams in a similar way. For example to refer to the variable T in stream S1 use:

```
S1.T
```

If you have a block and stream with the same name, or if you are editing an Aspen Dynamic simulation, you need to explicitly define whether you are referring to a variable within a block or a stream. To do this use the syntax:

```
BLOCKS(”BlockName”).VariableName
STREAMS(”BlockName”).VariableName
```

For example:

```
BLOCKS(”HX1”).TOut
STREAMS(”S1”).T
```

If a block is within a hierarchy block you also need to include the hierarchy block name. For example if HX1 is in the hierarchy block RefrigPlant, then refer to Tout as RefrigPlant.HX1.Tout.

Flowsheet Constraint Examples

The following example shows a flowsheet constraint equation which equates the duty for two heat exchangers in the flowsheet.

```
CONSTRAINT
 HX1.Q = -HX2.Q;
END
```

The next example shows how to specify values for the ambient temperature variables within three blocks in the flowsheet.

```
CONSTRAINT
 C1.Tamb = 20, Fixed;
 C2.Tamb = 20, Fixed;
 C3.Tamb = 20, Fixed;
END
```

The last example shows how to declare a new flowsheet variable which represents ambient temperature, and equates the ambient temperature to the local values within blocks in the flowsheet.

```
CONSTRAINT
 Tamb as Temperature;
 Tamb: 20, Fixed;
 C1.Tamb = Tamb;
 C2.Tamb = Tamb;
 C3.Tamb = Tamb;
END
```

Defining Optimization Constraints

Optimization is used to manipulate decision variables to calculate the maximum or minimum possible value of an objective function. Typically you

need to apply constraints to the optimization to ensure that the solution meets product specifications, and is within safe and practical operating limits of the process.

Constraints can be defined within the Optimization tool graphical interface, and this approach is recommended for new simulations. The ability to define constraints in input language has been retained for backwards compatibility, and is documented in this section.

You can define constraints for your optimization simulation in the Flowsheet Constraints section. You can define two types of optimization constraints:

Type of constraint	Used for
Inequality	Ensuring the optimization solution is within process limits. For example pressure is below maximum working pressure, impurity levels are below maximum allowed, and so on.
Equality	Defining values that must be exactly satisfied when performing an optimization run.

Note: An equality constraint is different from an equation in the Constraints section because:

- An equality constraint does not reduce the degrees of freedom.
- An equality constraint is only active during optimization simulations. In other runs modes it will be ignored.

When using constraints you should take care to ensure that there is a feasible solution within the constraints you specify. This is particularly important when using equality constraints.

When performing a dynamic optimization the constraints defined in the Flowsheets Constraints section must only be satisfied at the end of the dynamic run. They are not applied at other times during the dynamic run. In other words, they are end point constraints.

Syntax for Constraints

```
ConstraintName: Expression1 ComparisonOperator Expression2;
```

ConstraintName ..... Optional name for identifying the constraint

Expression ..... An expression

ComparisonOperator ..... One of:

<= -less than or equal to

>= - greater than or equal to

== - equals

It is recommended that you name your constraint equations so that you can easily identify constraints when the results are reported.

Examples of Defining Constraints

The following example shows a Flowsheet Constraints definition that contains a number of inequality constraints.

```
MAX_EFFLUENT_CONC as RealVariable(fixed, 1E-3);
MAX_TEMPERATURE   as RealVariable(fixed, 124);
MIN_TEMPERATURE   as RealVariable(fixed, 20);
MAX_PRESSURE      as RealVariable(fixed, 6.3);
MIN_PRESSURE      as RealVariable(fixed, 1.0);

Conc:  PURGE.X1("Methanol") <= MAX_EFFLUENT_CONC;
MinTemp: MEREACTOR.TEMP     >= MIN_TEMPERATURE;
MaxTemp: MEREACTOR.TEMP     <= MAX_TEMPERATURE;
MinPres: MEREACTOR.PRESSURE >= MIN_PRESSURE;
MaxPres: MEREACTOR.PRESSURE <= MAX_PRESSURE;
```

Defining Steady State Constraints

You can define constraints for your steady state optimization simulation directly in the Steady State Constraints tab in the Optimization Tool dialog.

To specify a constraint on a free variable:

1. Open the Optimization dialog box make sure Steady State Optimization is selected in the Setup tab.
2. Click the Steady State Constraints tab.
3. To add variable names to the grid, do one of the following:

   Type a variable name in the Variable text box and then click the Add button.

   Drag and drop variable names from forms onto the grid.

   Click Find, and drag and drop variable names from Variable Find onto the grid.
4. Edit the upper and lower bound values to the values you wish the variable to be constrained between during the steady state optimization simulation. These bounds are the same as the variable bounds by default.

Notes:

- Constraints in the Steady State Constraints grid are applied *only* for steady state optimization simulations. They are not active in any other simulation.
- They do not change the degrees of freedom of your problem.
- Use the Active switch to conveniently change between different constraints. Constraints are only active if the Active box is ticked.
- Use the Remove or Remove All buttons to remove one or all of your constraints.

Defining Dynamic Constraints

You can define constraints for your dynamic optimization simulation directly in the Dynamic Constraints tab in the Optimization Tool dialog. There are two types of dynamic constraint you can specify:

Final Time	The Constraint is applied at the final time of the simulation only.
Path	The Constraint can be applied at a finite number of points over the dynamic optimization time horizon.

To specify a constraint on a free variable:

1 Open the Optimization dialog box ensure Dynamic Optimization is selected in the Setup tab.

2 Click the Dynamic Constraints tab.

3 To add variable names to the grid, do one of the following:

Type a variable name in the Variable text box and click **Add**.

Drag and drop variable names from forms onto the grid.

Click Find, and drag and drop variable names from Variable Find onto the grid.

4 Choose the constraint type (Final time or Path) in the Constraint Type column.

5 If you choose Final Time, press the Edit button to define the upper and lower bounds you wish the variable to be constrained to at the final time. These bounds are the same as the variable bounds by default.

6 If you choose Path, press the Edit button to define the Path Constraints.>

Notes:

- Constraints in the Dynamic Constraints grid are applied *only*

for dynamic optimization simulations. They are not active in any other simulation.

- They do not change the degrees of freedom of your problem.
- Use the Active switch to conveniently change between different active constraints.
- Use the Remove or Remove All buttons to remove one or all of your constraints.
- The values of the variables at each constraint plus the corresponding Lagrange Multipliers after a successful dynamic optimization are available in the Final Time Constraint and Interior Point grids (available from the Edit button).
- At most you can have one final time and one path constraint active on the same variable.

Path Constraints

These are constraints on free variables at a set of finite *interior points* in time over the time horizon of your dynamic optimization simulation. They can be used to discretize constraints on variables active over *all* time into a series of *finite* interior point constraints. For an example, refer to the Dynamic Optimization Example.

To define Path Constraints for a dynamic optimization simulation:

1 Select "Path" in the Constraint Type column of the Dynamic Constraints grid for the variable you wish to define a path constraint.

2 Click **Edit**.

3 In the Interior Points dialog, choose how many interior points per element you want to use. This is the finite number of points per element at which you wish to constrain the variable. The points are spread evenly through each element, for example: if you choose 1, the constraint is at the end of each element; if you choose 2, the constraint is at 0.5 through the element and at the end of the element, and so on.

4 In the Interior Points dialog, specify the upper and lower bounds you want to constrain the variable to at each interior point.

5 Use the Active switch to define which elements and interior points you want to constrain the variable. For example, you may only wish to constrain variable during the first few elements of the simulation.

Notes

- The default values of the upper and lower bounds are the variables upper and lower bounds.

- If you define some constraints and then change the number of interior points per element, the existing interior point constraints will be re-sampled to the new number of interior points, thus avoiding re-entering data.

3 Modeling Language for Types

This chapter describes how to define the following types:

- Variables
- Parameters
- Ports
- Streams
- Properties
- Procedures

Definition of Models is covered in Chapter 4.

Defining Variable Types

Aspen Custom Modeler includes a large number of pre-defined variable types. To see these, in Simulation Explorer look in the Variable types folder of the Modeler library. If an existing variable type meets your needs we recommend that you use this. However for some applications you may need to define additional variable types.

Use the VARIABLE keyword to define variable types. Once a variable type is defined you can then create variables that use that type, for example within models. These variables inherit their default property values from the variable type. Variable types are a convenient way of defining variables with common properties, and normally correspond to physical quantities such as pressure, flow rate and so on.

Variable Type Definition Syntax

```
VARIABLE VariableType USES InheritedVariableType
   Property1: PropertyValue1;
   Property2: PropertyValue2;
      :
END
```

VariableName	The name of the new variable type
InheritedVariableName.....	The optional name of an existing variable type from which the new

	variable type will inherit its properties
Property	A property whose default value you wish to change from that in the inherited type
PropertyValue	The default value for the property

If you do not inherit from an existing variable type, the new variable type will inherit its property values from the built in RealVariable variable type. For information on RealVariable, see Chapter 1.

To change a property value from that in the inherited type, assign a value to that property in the variable type definition.

Properties which you can change are:

Value	The value of the variable.
Upper	The upper bound for the value. Default is 1.0e37.
Lower	The lower bound for the value. Default is –1.0e37.
Scale	The scaling factor used by the solver. Default is 1.0.
PhysicalQuantity	The name of the physical quantity definition used by the variable.
Description	A text string that can be used to store a description of the variable.
Units	Current unit of measurement used to display the value.
Tag	Label for use in interfaces to online applications.
Record	Controls whether the time history of the variable value is always recorded. Can be True or False. Default is False.
Spec	The specification for the variable. Can be Free, Fixed, Initial, RateInitial. Default is Free.

You can also add properties of type Integer, Real, Logical or String to your own variable types. These added properties can only be accessed through the automation interface e.g. from scripts or an external Visual Basic application. These can be useful if you need to track some additional variable property in your external application.

VARIABLE Examples

The first example shows the definition of the variable type Temperature:

```
VARIABLE Temperature
  PhysicalQuantity: "Temperature";
  Value: 25;
  Lower: -246;
  Upper: 5000;
```

```
END
```

The next example shows the definition of a variable type for criogenic temperatures. This inherits from temperature, but has a different default value and upper and lower bounds.

```
VARIABLE CrioTemperature
  PhysicalQuantity: "Temperature";
  Value: -100;
  Lower: -270;
  Upper: 0; END
```

Defining Parameter Types

You can define new parameter types. This is most useful for creating your own StringParameter types with lists of valid values. When a parameter of this type is displayed in a table the valid values will then automatically appear on drop down selection lists for the parameter value.

Parameter Type Definition Syntax

```
PARAMETER ParameterType USES BuiltInType
  VALUE: DefaultValue;
  VALID AS SetType (ValidList);
END
```

ParameterType	The name of the new parameter type.
BuiltInType	One of the built-in parameter types: RealParameter. IntegerParameter. LogicalParameter. StringParameter.
DefaultValue	The default value of the parameter type.
SetType	Can be either INTEGERSET (if *BuiltInType*=IntegerParameter) or STRINGSET (if *BuiltInType*=StringParameter).
ValidList	A list of valid values for this parameter type. Can be either a list of integers (if *BuiltInType*=IntegerParameter) or text strings (if *BuiltInType*=StringParameter).

A list of valid values can only be supplied for parameter types that inherit from IntegerParameter or StringParameter. This list is optional.

Parameter Type Definition Example

The following example shows how to define and use a new parameter type that is used to define the geometry of a vessel.

```
PARAMETER GeometryParameter USES StringParameter
 Valid AS StringSet(["Cylinder", "Cone",
  "Hemisphere", "Rectangular"]);
 Value: "Cylinder";
END

MODEL Tank
 Geometry AS GeometryParameter("Cone");
  :
 IF Geometry == "Cone" THEN
  Volume = PI/3 * Radius * Radius * Height;
 ENDIF
 IF Geometry == "Rectangular" THEN
  Volume = Length * Length * Height;
 ENDIF
END
```

Once the model is instanced as a block you can change the parameter Geometry from the AllVariables Table for that block. The table will show a drop down list of the value values.

Defining Port Types

Use port types to define what variables will be passed in or out of a model in streams connected to the model. For each variable you define its name and variable type.

PORT Syntax

```
PORT PortType USES InheritedPortType
  PortVariableName1 AS VariableType1;
  PortVariableName2 AS VariableType2;
    :
END
```

PortType	The name of the port type
InheritedPortType.....	Optional. The name of an existing port type from which the port type will inherit properties.
PortVariableName.....	The name of a variable to be passed through this port

VariableType.....	The variable type for a variable passed through the port

PORT Remarks

You can optionally make the port type inherit the properties of a previously defined port with the keyword USES. You can then add additional variable types to the existing definition so that more information is passed by the new port type.

PORT Examples

The following example shows the definition of the port type Material. This passes a molar flow rate and a temperature:

```
PORT Material
  F AS Flow_mol;
  T AS Temperature;
END
```

The next example shows that the definition of port type NewMaterial inherits the properties of the previously defined type Material. The port variable P is added to the list of values communicated by the port:

```
PORT NewMaterial USES Material
  P AS Pressure;
END
```

Defining Stream Types

To simply connect variables between an output port in one model and an input port in another, you can use the built in Connection stream type. You can create your own stream types to create more intelligent stream that can contains additional variables, equations, forms and so on.

Stream types are introduced by the keyword STREAM. A stream is similar in syntax to a model.

STREAM Syntax

```
PRIVATE STREAM StreamType USES InheritedStreamType
  StreamStatements
END
```

PRIVATE	Optional keyword that makes the model text inaccessible to the user when the model is being used from within a library.
StreamType.....	The name of the stream type.
InheritedStreamType.....	Optional. The name of a stream

	type from which this stream type will inherit properties.
StreamStatements	A list of statements that define the stream type.

STREAM Remarks

The statements you can use in a stream type definition are the same as those in a model definition, except that:

- You must define at least one port, but not more than two ports. Most streams connect two blocks together, and therefore need two ports. You can choose to define special stream types for feeds or products that have only one port.
- If you define two ports, one must be an INPUT port, and the other an OUTPUT port.

If you use the optional PRIVATE keyword and create a library that includes the stream type, the user of the library cannot:

- See the stream type in the Simulation Explorer view.
- Export the stream type with a Save.
- See the stream type equations and variable names in a Diagnostic view.
- PRIVATE can be used to protect proprietary information contained within the stream type definition.

STREAM Example

In this example, a stream is defined as having a potential loss of fluid through a leak. By default the leak flow is fixed and has a value of zero.

```
STREAM LeakyStream
 FLeak        AS Flow_mol (Fixed, 0);
 StreamInput  AS INPUT  Material;
 StreamOutput AS OUTPUT Material;
 StreamInput.F = StreamOutput.F - FLeak;
END
```

Defining External Procedures

You can use procedure types to interface your models to external Fortran, C, or C++ routines. These routines can range from simple utility routines to physical properties calculation, or complete legacy models.

This section explains how to link your models to procedures. For information on writing procedures and functions, see the online Help *Improving Your Simulations*, Designing Models, Using Procedures and Functions.

Procedure Syntax

```
PROCEDURE ProcedureName
  CALL: "RoutineName";
```

```
  LIBRARY: "dllname";
  INPUTS: InputArgumentType, InputArgumentType, ... ;
  OUTPUTS: OutputArgumentType, OutputArgumentType, ... ;
  OPTIONS: OptionsList;
  WORKSPACE: WorkspaceExpression ;
  LANGUAGE: "FORTRAN | C | C++";
  COMPATIBILITY: "SPEEDUP";

  IMPLEMENTATION: SUBROUTINE | FUNCTION "SourceName" |
  TEXT

   SourceCode

  ENDTEXT;

END
```

ProcedureName	The name of the procedure
RoutineName	The name of the routine in the external code. If the RoutineName is the same as the ProcedureName, the CALL statement is not required
dllname	The name of the dll containing the compiled procedure code. You need to specify the full path unless the dll is located: - in a folder that is on the PATH; - in the simulation Working Folder, or; - in the same directory as the Input language text file. You do not need to include the extension **.dll**. A single dll may contain routines for more than one procedure.
InputArgumentType	The types in the input argument list. Can be any variable type, RealParameter, IntegerParameter, or StringParameter.
OutputArgumentType	The type of a variable in the output argument list. Can be any variable type.
OptionsList	Optional. A list of the options used for the procedure.
WorkspaceExpression	Optional. Either a fixed integer value for workspace, or an integer expression (literal, that is, numbers not symbols, constants only, for example, 10*(10-1)/2).
SourceName	Optional. The full pathname of the file containing the source code for the procedure. You do not need to specify the file type extension.
SourceCode	Optional. The FORTRAN, C or C++ source code for the procedure.

The INPUT and OUTPUT argument lists define the types of parameters and variables that will be passed to and from the procedure when it is used in a model or a stream type. This information is used to check that you are passing the correct information to and from the procedure. If you do not want to restrict the types of variables that can be used as an argument to or from a procedure, use the type RealVariable in the argument list. This will allow variables of any type to be passed as this argument.

In the case where an array is passed to the procedure, follow the variable type name with dummy subscripts, using one asterisk for each dimension, for example, (*) or (*,*).

The WORKSPACE statement can be a fixed integer number or integer expression which allocates that amount as workspace to be used by the external routine.

The LANGUAGE statement indicates whether the routine is written in Fortran, C, or C++. If there is no LANGUAGE statement the default of Fortran is assumed.

The IMPLEMENTATION statement indicates whether the external routine is a subroutine or a function. If no IMPLEMENTATION statement is present, the default is subroutine.

You can optionally provide either the full path to the file which contains the source code, or include the source code for the procedure between the keywords TEXT and ENDTEXT. If you select the Generate Code option for the procedure you can have ACM automatically compile this code and build your code into a dll ready for use in your simulation.

The COMPATIBILITY statement enables you to use Fortran code that was originally developed for use with SPEEDUP 5.5. This means that you do not need to alter the call list for SPEEDUP compatible code. However, Aspen Custom Modeler is stricter than SPEEDUP in enforcing the correct argument list for an external procedure call. If you have unresolved link symbols, check that you procedure argument list contains all of the required arguments. A common mistake is to miss out IFAIL or ITYP at the end of the argument list. The COMPATIBILITY statement can only be used when Fortran is used to implement external procedures.

The OPTIONS statement is a list of keywords that indicates extra options about the procedure call. The default is no extra options. The options available are:

Keyword	Meaning
PRECALL	The procedure is called before the run starts, in the following situations: • After loading a new simulation. • After changing the structure of a simulation. • Re-starting a simulation after an Interrupt.
POSTCALL	The procedure is called in the following situations: • Loading a new simulation. • Changing the structure of the simulation.

	• Using Interrupt during a simulation run.
DERIVATIVES	Indicates that you have chosen to calculate and return analytical derivatives of the procedure outputs with respect to the inputs.
PRESET	Indicates that the current values of the procedure outputs should be passed into the procedure when it is called. This is useful if wish to use one of these values in your procedure, for example as an initial estimate of the new output value to be calculated.
PROPERTIES	Indicates that this is a physical properties procedure, and that the component list identifier should be passed as an input argument so that the routine can determine the component name and property calculation options to use.
CHANGESINPUTS	Enables you to change the value of an input variable in the external code. You can use this to calculate and return an initial estimate for a variable. Use this option only if the procedure is always torn; using this feature may cause the solver to fail to converge if the procedure is not torn.
ALWAYSCALL	By default the solver detects whether the inputs to a procedure have changed since it was last called, and if not it skips the call to the procedure. This option forces a call to the procedure even when the inputs (and, if PRESET is also specified, the outputs), have not changed since the last call. **Note:** A procedure with no inputs behaves as if the ALWAYSCALL option has been specified.

External Procedure Example

The following example shows a call to a Fortran subroutine that calculates the volume of a vessel, given the height and cross sectional area:

```
PROCEDURE VolCalc
  CALL    : "vc";
  LIBRARY : "C:\Examples\mlp";
  INPUTS  : Length, Area;
  OUTPUTS : Volume;
  OPTIONS : DERIVATIVES;
  LANGUAGE: "Fortran";
  IMPLEMENTATION : SUBROUTINE "C:\Examples\volcal";
END
```

Defining Structure Types

Structures are designed to support the definition of, access to and persistence of global data to be used with a simulation. Typically they will contain variables and parameters the values of which can then be referenced from model and stream instances.

STRUCTURE Syntax

```
PRIVATE STRUCTURE StructureName USES StructureName
StructureDescriptionStatements
END
```

PRIVATE	Optional keyword that makes the structure type text inaccessible to the user when used in a library. This can be used to protect proprietary information contained within the structure type.
StructureName	The name you give this structure type description

STRUCTURE Remarks

You can include statements in the structure description to define different aspects of the structure. Structures do not need to contain specific elements. For example, you can define a model with no equations and use inheritance to define different structures with different sets of global data.

You can add the following statements to your `structure` definition:

- Parameters.
- Sets.
- Variables.
- Ports.
- Equations.
- Procedure calls.
- Sub-structures (instances of other structures).
- Forms and scripts associated with this structure type.

You can mark the structure as PRIVATE. This means that if the structure is used in a library, the user of the library cannot:

- Display the structure type in the Simulation Explorer view.
- Export the structure type with a Save.
- See the structure type equations and variable names in a Diagnostic view.

Example

EXTERNAL qualifier.

Using EXTERNAL to reference structure instances.

You can use the EXTERNAL keyword to define a reference to an instance of a structure type from which global data can be obtained.

e.g. Rxn1 as External Powerlaw;

where Powerlaw is the name of a structure type and Rxn1 is the reference you will be using in this model. Having defined this reference you can use it to access parameters and variables in the structure type.

e.g. Rxn1.HeatofReaction.

This could then be used wherever local parameters or variables are used. However note you cannot assign a value to a parameter or variable in a structure from a reference. You can only do that in the structure type itself.

Example

```
Rxn1 as EXTERNAL Powerlaw;
```

4 Modeling Language for Models

Models describe the behavior of a unit operation or other item that you wish to include in your simulation flowsheet. This chapter describes how to define models. The following topics are covered:

- Model Syntax.
- Declaring variables.
- Declaring ports.
- Writing equations.
- Equations with array variables.
- Conditional equations.
- Using sets in conditional expressions.
- FOR loops and the ForEach operator.
- Equations that call procedures.
- Equations that refer to port variables.
- CONNECT keyword.
- LINK keyword.
- Connectivity rules.
- Using SIGMA.
- Using the SIZE function.
- Assigning specifications.
- Referring to sub-models.
- Using SWITCH.
- Using external properties to write sub-models.
- Using virtual types in sub-models.

MODEL Syntax

```
PRIVATE MODEL ModelName USES InheritedModel
  ModelStatements
END
```

PRIVATE	Optional keyword that makes the model text inaccessible to the user when the model is being used from within a library.
ModelName	The name of the model.
InheritedModel	Optional. The name of a model from which this model will inherit properties.
ModelStatements	A list of statements that define the model.

If you use the optional PRIVATE keyword and create a library that includes the model, the user of the library cannot:

- See the model in the Simulation Explorer view.
- Export the model with a Save.
- See the model equations and variable names in a Diagnostic view.

PRIVATE can be used to protect proprietary information contained within the model.

Model statements can include some or all of the following:

- Variable, Parameter and Set declarations.
- Port declarations.
- Equations.
- Procedure calls.
- Instances of other models (sub-models).
- Connections between sub-models.
- Links between sub-models and ports.
- Assignments to variable or parameter properties.

These are described in more detail in the following sections.

Declaring Variables in Models

Use variable declaration statements in models to define the variables you need for your model equations. You can also define array variables with one or more dimensions.

You can define a variable to be measured or manipulated by defining a direction for the variable. This means you can connect two variables in different blocks on the flowsheet using the ControlSignal stream.

You can also define a variable as hidden, so that by default the variable is not seen by the model user.

Model Variable Syntax

```
VariableName1(Set1 , Set2, ...), VariableName2(Set, …), ...
AS FlagList VariableType (SpecificationList);
```

VariableName.....	The name of the variable being declared
S	Optional index set definition used to declare the index set for array variables. Either a set name or a set definition between **[** and **]** (brackets). To define multi-dimensional arrays, define one set for each dimension.
FlagList	Optional. List of optional qualifiers for the variable. Can include a Direction (INPUT, OUTPUT), HIDDEN, UseUOMof, or GLOBAL.

INPUT, OUTPUT	Direction qualifiers that specify that a variable will be available as a control input (manipulated variable) or control output (measured variable). You can use both keywords to make a variable available as both an input and an output. For information on the rules for connecting variables with a stream, see CONNECT Keyword on page 4-107.
HIDDEN	An optional qualifier that defines a variable that by default is not displayed in Tables. Use HIDDEN for variables that are likely to be of little interest to the end user of the model. HIDDEN variables can still be accessed through the automation interface.
EXTERNAL	Indicates that the declared property is not part of the model but instead is a reference to a property of the containing model.

GLOBAL	Indicates that the declared variable or parameter is not part of the model but instead is a reference to a global variable. For example, a global parameter could be the Universal Gas constant.
VariableType.....	The variable type to be used for the variables.
SpecificationList.....	Optional list of specifications for the variables.

Model Variable Remarks

You can declare a list of variables with the same type in one statement. Each variable inherits all of the property values for it's variable type.

Model Variable Examples

In the first example, a variable called TFeed is declared as being of the generic built in variable type RealVariable:

```
FeedTemp AS RealVariable;
```

In the second example, the variable TFeed is declared to be of variable type Temperature, and it's default spec is defined as Fixed:

```
TFeed AS Temperature (Fixed);
```

In the next example, TFeed is declared to be of variable type Temperature. The default value and the lower bound are specified and override those inherited from variable type Temperature:

```
TFeed AS Temperature (Value:373.0, Lower:273.0 );
```

In the next example, no specification keyword has been used. This means the number in brackets is assigned to the value property of the variable. The direction keyword is included in the statement, which means the value of TFeed can be passed to another block on the flowsheet using the built-in stream type ControlSignal.

```
TFeed AS OUTPUT Temperature (373.0);
```

The following example shows two variables defined in one statement. Both variables have the default value and the lower bound altered from the values inherited from variable type Temperature.

```
TFeed, TProd AS Temperature (373.0, Lower:273.0);
```

In the following example, the variable FlowIn is defined as a four element, one-dimensional array, and FlowOut is defined as a three element one-dimensional array of variable type MolFlow.

```
Nsect AS IntegerParameter(4);

Index AS IntegerSet([1:Nsect]);

FlowIn(Index), FlowOut([1:Nsect-1]) AS MolFlow;
```

A two-dimensional array variable is declared in the following example:

```
NPoints  AS IntegerParameter(20);

Points   AS IntegerSet([1:NPoints]);

Elements AS IntegerSet([1:30]);
```

BulkTemperature(Points, Elements) AS Temperature;

The following example shows how to declare hidden, input and output variables. h is hidden and by default is not shown in tables or plots for the model. T is a control output, and so can be used as a measured variable in a control scheme. Duty is both a control input control output, and so can be used as a measured and/or manipulated variable in a control scheme.

```
h    AS HIDDEN Enth_mol;

T    AS OUTPUT Temperature;
```

Duty AS INPUT, OUTPUT Enthflow;

This example shows how to reference the global variable TAmbient. The statement below can be included in one or more models. As soon as any one of these models is instanced TAmbient will be instanced. As further models are instanced they will all share the same global TAmbient. This means that you can change the value of TAmbient on the Simulation Globals table, and this will affect all of the models that reference it. This is useful for quantities such as ambient temperature, where you may want to use the same value across all of your simulation.

```
TAmbient AS GLOBAL Temperature;
```

The last example shows how to define a variable to use the units of measurement of another variable. You can also see this feature in use in the PID controller model in the Modeler library.

If the variable, PV, is connected to a level variable in a tank, using a ControlSignal stream, then PV will automatically be displayed in the units of measurement of the level variable. The UseUOMof keyword is used to also ensure that SPRemote will be displayed in the units of measurement of the level variable.

```
PV AS Input Control_Signal (Description:"Process
variable");

SPRemote AS Input Control_Signal (Fixed,
Description:"Remote
```

```
  setpoint", UseUOMof:"PV");
```

Declaring Ports in Models

Use ports to define what variables and parameters the model can pass to or from other models via streams. Ports can be connected to graphically within the flowsheet, or by using the CONNECT and LINK keywords within a model.

Model Port Syntax

```
PortName AS DirectionType PortType ;
```

PortName.....	The name you give to the port.
DirectionType.....	Can be either INPUT or OUTPUT.
PortType	The type of port. A definition for this port type must already exist with Custom Modeling or an open library.

Model Port Remarks

The port name is the name you provide for the port. This name is used to refer to the port variables related to the port type. You define the names of the port variables in the port type declaration statement.

Model Port Example

In this example, the ports Input1 and Output 1 are defined as using the port type Material. Input1 has direction INPUT, and Output1 has direction OUTPUT. An equation that equates the inlet and outlet flow rates is shown below. The port type definition for port type Material is also shown.

```
PORT Material
 Flow AS Flow_Mol_Liq;
 x(ComponentList) AS Molefraction;
 T as Temperature;
END


Model Heater
   :
 Input1  AS INPUT  Material;
 Output1 AS OUTPUT Material;
 Input1.Flow = Output1.Flow;
   :
End
```

Using Port Properties

A port has properties that you can use in the model to determine how the port is used. These are:

Property	Description
IsConnected	A logical parameter that indicates whether a stream is connected to the port.
ComponentList	The set of components being used in this port.

Using Port Properties Examples

This example shows how the variable Output1.flow is calculated, dependant upon the value of the IsConnected parameter.

If there is no connection to the port Offtake, Offtake.IsConnected has a value of False, and Input1.flow is equated to output1.flow. If the port Offtake is connected, then the offtake flow is included in the equation.

```
MODEL BLEED

 Input1  AS INPUT  Material;
 Output1 AS OUTPUT Material;
 Offtake AS OUTPUT Material;

 IF NOT Offtake.IsConnected THEN
  Input1.flow = Output1.flow;
 ELSE
  Input1.flow - Offtake.flow = Output1.flow;
 ENDIF

END
```

The following example shows how to use two different port component lists:

```
PORT Process
 Flow AS Flow_Mol_Liq;
 x(ComponentList) AS Molefraction;
END

MODEL ComponentMixer

 Quench  AS INPUT  Process(ComponentList: AqueousFluid);
```

```
 Inlet1  AS INPUT  Process(ComponentList: ProcessFluid);
 Outlet1 AS OUTPUT Process(ComponentList: AllComponents);
   :
 /* Loop on components only in Inlet1 port */
 FOR I IN [Inlet1.ComponentList - Quench.ComponentList] DO
  Outlet1.Flow*Outlet1.x(i) = Inlet1.Flow*Inlet1.x(i) +
   Quench.Flow*Quench.x(i);
 ENDFOR

 /* Loop on components only in Quench port */
 FOR I IN [Quench.ComponentList - Inlet1.ComponentList] DO
  Outlet1.Flow*Outlet1.x(i) = Inlet1.Flow*Inlet1.x(i) +
   Quench.Flow*Quench.x(i);
 ENDFOR

 /* Loop on components only in both ports */
 FOR I IN [Quench.ComponentList * Inlet1.ComponentList] DO
  Outlet1.Flow*Outlet1.x(i) = Inlet1.Flow*Inlet1.x(i) +
   Quench.Flow*Quench.x(i);
 ENDFOR
   :
END
```

The three FOR loops are designed to catch every situation for the components entering the mixing vessel, whether there are duplicate components in the two ports or all components are unique in both ports.

Using the IsMatched Attribute of Variables

You can connect a port to a port that does not have completely matching variables, for example, the component list used in each port may be different. You can test for this by checking the IsMatched attribute of the variables in ports.

When a port is connected, each variable that is matched in the connected port has the IsMatched attribute set to True. If it was not matched, IsMatched will be False.

Note: If a port is not connected, the value of IsMatched is undefined for the variables in that port.

undefined for the variables in that port.

Example of Using IsMatched

The following example uses the IsMatched attribute in a Mixer model that combines streams with different component sets. If a component is present in the inlet port, but is not matched in the stream connected to this port, the flow rate of that component is equated to zero.

```
PORT FlowPort
 F(componentlist) AS flow_mass;
END


MODEL ddd
 Inp AS INPUT MULTIPORT OF FlowPort;
 Out AS OUTPUT FlowPort;


 FOR c in Inp.componentlist DO
  IF (NOT Inp.F(c).IsMatched) THEN
   Inp.F(c) = 0.0;
  ENDIF
 ENDFOR


END
```

Using Multiports

You can define a collection of ports, which is known as a multiport. A multiport allows multiple stream connections to be connected to it. A multiport can use the same port types as a normal port.

Multiport Syntax

The syntax for defining a multiport in a model is:

```
PortName AS DirectionType MULTIPORT OF PortType
(MIN_CONNECTIONS: mincon, MAX_CONNECTIONS: maxcon);
```

PortName	The name you give to the port.
DirectionType	Can take the values INPUT, OUTPUT, or both.
PortType	The type of port. A definition for this port type must already exist within Custom Modeling or an open library.
mincon	Optional. The minimum number of

	connections allowed by this port. Must be an integer value of 0 or larger. Default is 0.
maxcon	Optional. The maximum integer number of connections allowed by this port. Must be an integer value of 1 or larger. By default there is no limit on the number of connections.

Multiport Remarks

A multiport contains additional properties about the current connections to it. You can use these in your model equations. The additional properties are:

ConnectionSet	A set of text strings which contain the names of the streams connected to the multiport.
Connection	An array of the connected ports which is indexed over ConnectionSet.

For information on accessing the properties of a multiport, see Using Connection with Multiports on page 4-92.

Using Connection with Multiports

You use Connection to access port variables in a multiport.

Connection Syntax

The variables in the port are accessed using the following syntax:

```
PortName.CONNECTION(StreamName).PortVariable
```

As Connection is an array, you can use the SIGMA function to sum the values of PortVariable for all connections made to that port. If the PortVariable is also an array, you can use SIGMA in the following three ways.

The first way sums all combinations of Connection(i).ArrayVariable(j).

```
Sum = SIGMA(PortName.CONNECTION.ArrayVariable);
```

For a detailed explanation, see:

SIGMA for Multidimensional Arrays Examples.

Using SIGMA a second way, for each element of the array variable, the values of that ArrayVariable in all connections made to the multiport are summed:

```
FOR i IN ArrayIndexSet DO
  Sum(i) = SIGMA(PortName.CONNECTION.ArrayVariable(i));
ENDFOR
```

Using SIGMA a third way , for each connection made to the multiport, the values of all the elements of the ArrayVariable are summed:

```
FOR i IN PortName.ConnectionSet DO
  Sum(i) = SIGMA(PortName.CONNECTION(i).ArrayVariable);
ENDFOR
```

You can use ConnectionSet to size an array variable to the same size as the number of connections to the multiport:

```
Variable (Portname.CONNECTIONSET) AS VariableType;
```

Multiport Example

The following model represents a stream mixer which can be used to mix up to 10 streams into a single outlet stream. The first equation sums the values of Flow across all connections to Input1. The FOR loop then performs a mass balance for each component to calculate the outlet composition. The sigma function sums the product of flow and fraction of a component across all connections.

```
PORT Main
 Flow AS Flowrate;
 Z(ComponentList) AS Fraction;
END


MODEL MULTIMIX


 Input 1 as INPUT MULTIPORT OF
  Main (MIN_CONNECTIONS:1, MAX_CONNECTIONS:10);
 Output1 AS OUTPUT Main;


 Output1.Flow = SIGMA(Input1.Connection.Flow);


 FOR i IN ComponentList DO
  Output1.z(i)*Output1.Flow =
 SIGMA(Input1.Connection.z(i)*Input1.Connection.Flow);
 ENDFOR


END
```

Note that in this example, if there are three streams S1, S2 and S3 connected to the port Input, and each of these carries water only, then the equations produced by this model are:

```
Output1.Flow = Input1.Connection("S1").Flow +
Input1.Connection("S2").Flow +
Input1.Connection("S3").Flow;
```

```
Output1.Flow*Output1.z("Water") =
Input1.Connection("S1").Flow*Input1.Connection("S1").z("Wat
er") +
Input1.Connection("S2").Flow*Input1.Connection("S2").z("Wat
er") +
Input1.Connection("S3").Flow*Input1.Connection("S3").z("Wat
er");
```

Writing Equations in Models

Use equations to define relationships between variables and parameters.

Equation Syntax

EquationName: ExpressionA = ExpressionB;

EquationName..... An optional equation identifier which is displayed in diagnostic output.

ExpressionA, ExpressionB..... Mathematical expressions involving variables and parameters.

Equation Remarks

You can have any number of variables on either side of the equality sign. You should arrange equations to improve the numerical solution of the simulation for robustness and speed, for example avoid dividing by a variable whose value may go to zero during solution.

The equations and variables in a model must completely describe the process. You must ensure that the simulation can be solved, given the number of known values in your simulation. The number of equations must equal the number of unknowns in your simulation.

Mathematical Operators in Equations

You can use the following mathematical operators in equations and assignments:

Operator	Usage	Description
+	x + y	Addition
-	x - y	Subtraction
-	-x	Negation
*	x * y	Multiplication
/	x / y	Division
^	x ^ n	Raised to the power of
SIN	SIN(x)	Sine
SINH	SINH(x)	Hyperbolic sine

COS	COS(x)	Cosine
COSH	COSH(x)	Hyperbolic cosine
TAN	TAN(x)	Tangent
TANH	TANH(x)	Hyperbolic tangent
ASIN	ASIN(x)	Arc sine
ACOS	ACOS(x)	Arc cosine
ATAN	ATAN(x)	Arc tangent
SQRT	SQRT(x)	Square root
SQR	SQR(x)	Square of
EXP	EXP(x)	Exponent
LOGe	LOGe(x)	Natural logarithm
LOG10	LOG10(x)	Logarithm to base 10
ABS	ABS(x)	Absolute value of
SIGMA	SIGMA(x,y,...)	Sum of values of variables and array elements
PRODUCT	PRODUCT(x,y,…)	Product of values of variables and array elements
MAX	MAX(x,y,…)	Maximum value in a list of arrays and variables
MIN	MIN(x,y,…)	Minimum value in a list of arrays and variables
TIME	TIME	Current simulation time
DELAY	DELAY x BY *lag*	Applies a time delay of time *lag* to variable *x*

TRUNCATE	TRUNCATE(x)	Removes the fractional part of floating point value x and gives the result as an integer. For example:
		`TRUNCATE(1.9)  ->  1`
		`TRUNCATE(2.1)  ->  2`
		`TRUNCATE(-1.9) -> -1`
		`TRUNCATE(-2.1) -> -2`

ROUND	ROUND(x)	Rounds the floating point value x to the nearest integer. For example:
		`ROUND(-1.9) -> -2`
		`ROUND(-1.5) -> -2`
		`ROUND(-1.3) -> -1`
		`ROUND(-1.0) -> -1`
		`ROUND(1.9)  ->  2`
		`ROUND(1.5)  ->  2`
		`ROUND(1.3)  ->  1`
		`ROUND(1.0)  ->  1`

Note: For trigonometric functions ensure that x has units of radians.

Equation Examples

The following is a simple flow continuity equation. FlowIn and FlowOut are variables defined in the model. The equation name MaterialBalance is optional.

```
MaterialBalance: FlowIn = FlowOut;
```

Input1.Flow and Output1.Flow are port variables. AdditionalFlow is a variable defined in the model. Both types of variable can be freely mixed.

```
Input1.Flow + AdditionalFlow = Output1.Flow;
```

Equations with Array Variables

Within equations and assignments you can refer to single array elements, a subset of the array elements, or to the full array.

To refer to a single array element use:

```
ArrayName(Index)
```

Where Index is the index of the required element.

To refer to a subset of the array use one of the following:

```
ArrayName ( [Index1, Index2, …] )
```

```
ArrayName ( IndexLower: IndexUpper] )
```

```
ArrayName ( [SetName] )
```

The first refers to the array elements with the specified index values. The second refers to the slice of elements with indices in the range from *IndexLower* to *IndexHigher*. This can only be used with arrays indexed over integer sets. The third refers to all elements whose indices are in the named set.

To refer to the full array use:

```
ArrayName
```

Array Variable Examples

In the following example, the statement `FlowIn = FlowOut;` creates four equations, one for each of the four array elements. This statement is automatically treated as an array equation because both the FlowIn and FlowOut arrays are indexed over the same set. SetFlow is defined as a scalar non-array variable.

```
MODEL Test
 NComps AS IntegerParameter(4);
 FlowIn([1:NComps])  AS Flow_mol;
 FlowOut([1:NComps]) AS Flow_mol;
 SetFlow             AS Flow_mol;
 FlowIn = FlowOut;
```

```
END
```

You can also have equations that involve single array elements, such as:

```
FlowIn(1) = FlowOut(1);
```

You can also have equations that operate on parts of an array. Both of the following are equivalent, and create three equations which equate elements 1, 2 and 3 of each array:

```
FlowIn([1,2,3]) = FlowOut([1,2,3]);
FlowIn([1:3])   = FlowOut([1:3]);
```

In the next example, because SetFlow is not an array, four equations are created which equate each element of FlowIn to SetFlow.

```
FlowIn = SetFlow;
```

The final example is legal, even though the number of elements on either side of the equation is not equal:

```
FlowIn([1, 3, 4]) = FlowOut;
```

This expression is interpreted as operating on the intersection of the array elements. Three equations are produced from this expression:

```
FlowIn(1) = FlowOut(1);
FlowIn(3) = FlowOut(3);
FlowIn(4) = FlowOut(4);
```

Conditional Equations

You can use conditionals for two purposes:

To switch between different equations during a simulation dependant upon the values of certain variables.

To make structural changes to the equations and/or assignments used depending upon the values of parameters or other aspects of the model configuration.

Conditionals are defined using the following syntax:

Run Time Conditional Equation Syntax

```
IF ConditionalExpression THEN
 Equation1A;
 Equation2A;
   :
ELSE
 Equation1B;
 Equation2B;
   :
ENDIF
```

Structural Conditional Syntax

```
IF ConditionalExpression THEN
 Equation1;
 Equation2;
   :
 Assignment1;
 Assignment2;
   :
ELSE
 Equation3;
 Equation4;
   :
 Assignment3;
 Assignment4;
   :
ENDIF
```

ConditionalExpression..... A conditional expression that evaluates to true or false.

Equationx..... An equation.

Assignmentx..... An assignment.

Conditional Equation Remarks

The conditional expression returns a value of either True or False. If the expression is true, the first set of equations and/or assignments are used. If the expression is false, the second set of equations and/or assignments are used.

For structural conditionals each branch of the conditional can take either a list of equations, a list of assignments, or both equations and assignments. The two branches can contain different numbers of equations. Also the Else branch is optional.

For run-time conditional equations, the two lists of equations must contain the same number of equations and the same variables, to keep the simulation square and complete. The else branch is always required.

For situations where there are a large number of branches in a run-time conditional, consider using a Switch statement instead of nested IFs.

You can use the following logical operators:

Operator	Meaning
== (two equals signs)	Equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

<>	Not equal to
AND	Logical AND function
OR	Logical OR function
NOT	Logical NOT function

Notes:

- The three logical functions are evaluated in the priority order: NOT, AND, OR.
- Two logical functions may not appear in sequence unless the second one is the NOT operator.
- To avoid ambiguities in your logical expressions, use parentheses to define the evaluation priority order.

Conditional Equation Example

The following run-time conditional equation represents a tank where both the inlet and outlet pipes are submerged below the liquid level. The conditional expression accounts for flow reversal.

```
IF Input.Press >= Output.Press THEN
  Input.Flow - Output.Flow = $Volume ;
  Output.Flow = K * SQRT (LiquidLevel) ;
ELSE
  Output.Flow - Input.Flow = $Volume ;
  Input.Flow = K * SQRT (LiquidLevel) ;
ENDIF
```

The next example shows a structural conditional based on a logical parameter whose value is TRUE:

```
IF IncludeValve THEN
  Output1.Pressure = Input1.Pressure -
                     Cv*Output1.Flow;
  Cv: 10.0, FIXED;
ELSE
  Output1.Pressure = Input1.Pressure;
ENDIF
```

The following example shows a structural conditional that switches between two equations dependant upon whether a feed is connected:

```
IF NOT Feed.IsConnected THEN
  $Holdup = Inlet.F – Outlet.F;
```

```
ELSE
  $Holdup = Inlet.F + Feed.F – Outlet.F;
ENDIF
```

In the next example is a structural conditional in which the not-equal-to operator, <>, is used to decide how pressure is calculated dependent on the value of the string parameter, ModeOfOperation.

```
IF ModeOfOperation <> "PressureDriven" THEN
  // ModeOfOperation has the value FlowDriven
  // Find minimum inlet pressure
  Pin = MIN(Inlet.Connection.P);
ELSE
  // ModeOfOperation has the value PressureDriven
  // Equate inlet pressures
  Inlet.Connection.P = Pin;
ENDIF
```

The following example shows a run-time conditional equation using the logical AND function to calculate the heat transfer area that is dependent on the level of liquid present within an evaporator.

```
IF level<level_low THEN
  HT_Area = 0;
ELSEIF (level>level_low AND level<=level_high) THEN
  HT_Area*(level_high–level_low) = MaxHT_Area*(level–
level_low);
ELSE
  HT_Area = MaxHT_Area;
ENDIF
```

Using Sets in Conditional Expressions

You can use sets in a conditional expression to test whether a parameter value is equal to one of the elements in a set.

Sets in Conditional Expressions Syntax

```
IF Expression IN SetExpression THEN
  :
ELSE
  :
```

```
ENDIF
```

Expression ..... Integer or String Expression

SetExpression ..... IntegerSet or StringSet

Sets in Conditional Expressions Remarks

The condition in this case can be interpreted as "if the Expression exists within the SetDefinition, then true, else false".

The Expression must be the same type as the base type of the set. Take care with operator precedence when the Expression is more than just a simple variable or parameter - parentheses may be needed for clarification.

Sets in Conditional Expressions Example

The following example tests whether the value of the string parameter KeyComponent is in the set of strings Components1.

```
Components1 AS StringSet(["CH4","C2H6"]);

IF KeyComponent IN Components1 THEN ...
```

The above statement is equivalent to:

```
IF (KeyComponent == "CH4") OR (KeyComponent == "C2H6") THEN
...
```

FOR Loops

FOR loops enable you to simplify repetitive statements. FOR loops are useful for handling arrays with large numbers of elements, or more than one dimension.

FOR Loop Syntax

```
FOR  Index  IN  SetExpression DO

  statements;

ENDFOR
```

Index ..... Index for this FOR loop.

SetExpression ..... An set expression.

statements..... A number of statements using the Index variable.

FOR Loop Remarks

The FOR loop is repeated with the index given each value in the set expression. The statement uses the index value to produce a different statement at each iteration. Note that there is no implied order for the iteration of the index, particularly when the set is a set of strings.

Statements in the FOR loop can be either equations or assignments.

FOR Loop Example

In the following example an equation and expression are repeated for all components in a component list.

```
MODEL Test


 FlowIn(ComponentList)  AS Flow_Mol;
 FlowOut(ComponentList) AS Flow_Mol;
 Split(ComponentList) AS Fraction;


 FOR I IN ComponentList DO
   FlowIn(I) = Split(I) * FlowOut(I);
   Split(I): 0.33, FIXED;
 ENDFOR


END
```

If ComponentList is defined as the set ["N2", "O2" and "CO2",] the index counter in the FOR loop, I, takes the value "N2", "O2" and "CO2", and three equations and three assignments are created.

Note that for this example, it is also possible to directly write a vector equation without using a FOR loop:

```
FlowIn = Split * FlowOut;
```

In this case, the number of equations created is based on the intersection of the sets defining the variables in the equation. As all three array variables are indexed by the same set, an equation is generated for each component.

The next example assigns a value of 0 to the flow rate of all components except water:

```
for i in Componentlist - ["H2O"] do
  F(i): 0;
endfor
```

The next example shows how to create a number of instances of a tray model within a column model, and how to simplify the repetitive CONNECT statements that connect the trays.

The loop repeats for all but the top and bottom stages in the column..

```
MODEL Column
 NStage AS IntegerParameter(20);
 StageSet AS IntegerSet([1:NStage]);
 Stage(StageSet) AS Tray;
```

```
 FOR J IN [2:NStage-1] DO
   CONNECT Stage(J).LiqOutFlow AND
           Stage(J+1).LiqInFlow;
   CONNECT Stage(J).VapOutFlow AND
           Stage(J-1).VapInFlow;
 ENDFOR
  :
END
```

In the previous example the number of stages, Nstage, is assigned a default value of 20 in the model. This value can be overwritten by changing the value on the AllVariables form for a block that uses the model.

ForEach Operator

The ForEach operator is the equivalent of the For operator, but is used only with list operators, that is:

- SIGMA
- PRODUCT
- MIN
- MAX
- UNION
- INTERSECTION
- DIFFERENCE
- SYMDIFF

List Operator Syntax

`ListOperator (Expression1, `*`Expression2`*`, …)`

ListOperator.....	Can be SIGMA, PRODUCT, MIN, MAX, UNION, INTERSECTION, DIFFERENCE, or SYMDIFF.
Expressionn	Any normal modeling language expression, for example, a variable name, a number, and so on. Can also be a ForEach expression, as shown in the following syntax.

ForEach Expression Syntax

`ForEach (Index1 in SetExpression1, `*`Index2 in SetExpression2`*`, …) Expression`

Indexn	**An index used in the expression.**
SetExpressionn	The set of values through which the index will be looped.
Expression	Must be valid type for the list operator, for example, if

the operator takes a list of set expressions then Expression must be a set expression.

ForEach Examples

The following is a simple matrix multiplication example:

```
Index([1:n]) as IntegerSet;

A(Index, Index), B(Index, Index), C(Index, Index) as
RealVariable;


For i in Index DO
  For j in Index DO
    C(i,j) = Sigma(ForEach (k in Index) A(i,k)*B(k,j));
  EndFor
EndFor
```

The result of the above for n = 2 would be the following equations

```
C(1,1) = sigma(A(1,1)*B(1,1), A(1,2)*B(2,1));

C(1,2) = sigma(A(1,1)*B(1,2), A(1,2)*B(2,2));

C(2,1) = sigma(A(2,1)*B(1,1), A(2,2)*B(2,1));

C(2,2) = sigma(A(2,1)*B(1,2), A(2,2)*B(2,2));
```

The following example equates the sum of the first three values in an array to zero:

```
sigma(foreach (i in set) x(i)) = 0;
```

When set = [1:3] this is equivalent to:

```
sigma(x(1), x(2), x(3)) = 0;
```

The second example sums array values across multiple sub-models:

```
sigma(foreach(i in set1, j in submodel(i).set)
model(i).x(j)) = 0;
```

When set1 = ["a", "b"], submodel("a").set = [1,2] and submodel("b").set = [3,4] , this is equivalent to::

```
Sigma(model("a").x(1),model("a").x(2),model("b").x(3),
      model("b").x(4)) = 0;
```

The third example sums values from two sets:

```
sigma(foreach(i in set1) x(i), foreach(i in set2) y(i)) =
0;
```

When set1 = ["a", "b"] and set2 = [1,2] the result is:

```
sigma(model("a").x(1),model("a").x(2),model("b").x(1),model
("b").x
(2)) = 0;
```

Equations that Call Procedures

You can use procedure calls in models to use an external subroutine to perform calculations. Before doing this the Procedure must be defined in a Procedure definition.

Procedure Call Syntax

```
CALL (OutputArgumentList)  = ProcedureName(
InputArgumentList ) ComponentList, TEAR;
```

ProcedureName	Name of the procedure to use.
InputArgumentList	List of input variables in the same order as the procedure definition.
OutputArgumentList	List of output variables in the same order as the procedure definition.
ComponentList	Optional name of a component list that applies to this individual call. The component and thermodynamic properties associated with this component list are used in property calculation calls. A value is required only when you want to override the default component list. For an example, see Using More Than One Component List in Models in Chapter 3.
TEAR	Optional. Specifies that this instance of the procedure is torn. For information on tearing, see the Aspen Modeler Reference, Chapter 6.

Procedure Call Remarks

The order of the arguments in the argument lists must be the same as that in the procedure definition. Also each argument must be of the same type as declared in the procedure definition, or of a type inherited from the declared type.

You can declare procedure calls at any point in the model definition after you have defined the variables used in the procedure call.

It is possible to use a slice of a multidimensional array as an argument to a procedure which expects a single dimension array. The single dimension array is build from the list of the elements specified from the multidimension array.

For example, the procedure pTestArray takes one single dimension array as input arguments:

```
Procedure pTestArray
  library: "testarray.dll";
  call: testarray;
  implementation: subroutine "testarray.f";
  language: "fortran";
  inputs: real(*);
```

```
  outputs: realvariable;
End
```

The model declares a two dimension array C. The first call will send the vector { C(1,1), C(1,2], C(1,3), C(1,4), C(1,5) }. The second call will send the vector { C(1,1), C(2, 1), C(3, 1) }. In other words, the procedure does not see the shape of the ACM array:

```
Model TestArray
  C([1:3], [1:5]) as realvariable (fixed);
  x as realvariable;
  x1 as realvariable;
  // a column of C
  call (x) = pTestArray (C(1, [1:5]));
  // a row of C
  call (x1) = pTestArray (C([1:3], 1));
End
```

Procedure Call Example

In the following example, VolCalc is the name of a procedure definition:

```
CALL (TankVolume) = VolCalc (TankHeight, TankArea);
```

The next example shows how to use the component list name after a procedure call to determine the components and properties used in the procedure calculation.

```
CALL (EnthOut) = Enthalpy (MolFracOut, PressOut, TempOut)
Process;
```

The enthalpy calculation is carried out using the component list Process. The component list determines the physical property calculation options to be used by this instance of the Enthalpy procedure.

Equations that Refer to Port Variables

Use port variables in equations to refer to the values being passed between blocks. Use the following syntax to refer to a port variable.

Port Variables Syntax

```
PortName.PortVariableName
```

PortName.....Name of the port in the model.

PortVariableName.....Name of the port variable in the port definition.

Port Variables Remarks

When you declare a port in a model, you can refer to any of the variables defined in the port's type definition.

Port Variable Example

This example defines a port type called Main, how this is instanced in a model, and how variables in this port are used.

```
PORT Main
  Flow  AS Flowrate;
  Temp  AS Temperature;
  Press AS Pressure;
END


MODEL SimpleValve


 // Define internal variable
 DeltaP AS Pressure ;


 /* Defined one inlet and one outlet port*/
 Input1 AS INPUT Main ;
 Output1 AS OUTPUT Main;


 // Equations - pressure change only
 Input1.Flow = Output1.Flow;
 Input1.Temp = Output1.Temp;
 Input1.Press - DeltaP = Output1.Press;


END
```

CONNECT Keyword

You use connection statements within a model to connect either ports of sub-models, or variables of sub-models . Ports are connected through streams.

For information on when to use CONNECT, and how to connect ports and variables, see Connectivity Rules.

CONNECT Syntax

Name: CONNECT BlockName.PortName AND BlockName.PortName *WITH StreamName;*

or

Name: CONNECT BlockName.VariableName AND BlockName.VariableName *WITH StreamName;*

Name	Optional name of the connection.
BlockName	Name of a sub model within the model.
PortName	Name of the port through which the connection is made.
VariableName	Name of a variable that has been defined as an Input or Output.
StreamName	Optional name of the stream to be used for the connection. For connecting variables, the stream must be of built-in type ControlSignal.

CONNECT Remarks

When connecting ports, one of the ports must be an input, and one of the ports must be an output. If you do not specify a stream, the ports are connected with a stream of an automatically generated name using the built-in stream type Connection. The Connection stream equates port variables of the same name and defines them as having equal values.

Caution: If you connect two ports of different types, there may be a mismatch in the variable list. The values that are passed over the connections are the intersection of the variables between the two ports. That is, the variables with names common to both ports are transmitted.

This means you can have fewer equations in your simulation than you expected, because the stream does not create an equivalence equation for all the variables in the ports. This usually makes the flowsheet under-specified. If you find your simulation is under-specified when you believe that the models are correctly defined, check for port mis-matching.

If you use the CONNECT statement to connect variables in a flowsheet, you must use the built-in stream type ControlSignal.

All input and output variables in sub-models are accessible in the containing model. You do not need to use LINK to make these accessible from the containing model.

CONNECT Restrictions

The following restrictions apply when using CONNECT:

- You cannot use a CONNECT statement to directly connect arrays of variables or ports. The following example is therefore illegal:

```
a([1:nStage]) as ModelType1;
b([1:nStage]) as ModelType2;
connect a.outlet and b.inlet; // Not valid !
```

- However, you can use a CONNECT statement to connect arrays of variables or ports using a FOR loop, as shown in the next example:

```
a([1:nStage]) as ModelType1;
b([1:nStage]) as ModelType2;
FOR i IN [1:nStage] DO
  connect a(i).outlet and b(i).inlet; // Valid
ENDFOR
```

- A CONNECT statement without a defined stream cannot appear in nested FOR loops where the index for any inner loop is dependent upon the index variable of an outer loop. This is because you cannot declare a suitable stream array to connect the ports or variables. The following example is therefore illegal:

```
nStage as IntegerParameter;
a([1:nStage]) as ModelType1;
b([1:nStage]) as ModelType2;

FOR i IN (1:a.nStage] DO
  FOR j IN [1:a(i).nArraySize] DO
    connect a(i).x(j) and b(i).y; // Not valid !
  ENDFOR
ENDFOR
```

CONNECT Examples

This example connects the ports Valve.Outlet and Tank.Inlet with a stream of type Connection, and gives this connection stream the name S1:

```
S1: CONNECT Valve.Outlet AND Tank.Inlet;
```

In the next example, the stream Stream1 is declared, and then used to connect Valve.Outlet and Tank.Inlet:

```
Stream1 as Connection;
CONNECT Valve.Outlet AND Tank.Inlet WITH Stream1;
```

The final example shows the output variable Tank1.Temp and the Input variable C101.PV connected with the built-in stream type ControlSignal:

```
CONNECT Tank1.Temp AND C101.PV;
```

LINK Keyword

Use the LINK keyword to define a connection to a containing model's ports from the ports of a submodel within the containing model. This exposes the sub-model's port as if it were a port of the containing model, and enables them to be connected to other models within a flowsheet.

The ports connected by LINK must always be of the same direction.

For information on when to use LINK, and how to connect ports, see Connectivity Rules.

LINK Syntax

```
Name: LINK PortName AND SMName.SMPortName;
```

Name	Optional name of the link.
PortName.....	Name of the port in the containing model.
SMName.....	Name of the sub-model instance within the containing model.
SMPortName	Name of the port in the sub-model.

LINK Remarks

Use the LINK keyword to connect ports in a sub-model to those in the containing model. Use the CONNECT keyword for connections between sub-models within the containing model.

LINK Restrictions

The following restrictions apply when using LINK:

- You cannot use a LINK statement to connect arrays of variables or ports. The following example is therefore illegal:

```
Feed ([1:nStage]) as input PortType1;
Stage([1:nStage]) as ModelType2;
link Feed and Stage.inlet; // illegal !
```

However, you can use a LINK statement to connect arrays of variables or ports in a FOR loop, as shown in the next example:

```
Feed ([1:nStage]) as input PortType1;
Stage([1:nStage]) as ModelType2;
for i in [1:nStage] do
  link Feed(i) and Stage(i).inlet;
endfor
```

- A LINK statement without a defined stream cannot appear in nested FOR loops where the index for any inner loop is dependent upon the index variable of an outer loop. The following example is illegal:

```
Feed ([1:nStage]) as input Multiport of Fluid;
Stage([1:nStage]) as StageModel;
for i in [1:nStage] do
 for j in Feed(i).Connectionset do
           // j is dependent upon i
  link Feed(i).Connection(j) and Stage(i).Outlet;
          // nested FOR loop too complex for LINK
 ENDFOR
ENDFOR
```

LINK Examples

In the following example, the LINK keyword is used to connect a port from the model Tank to a port of the model MultiTank. A number of Tank models are instanced within the model MultiTank. This method enables you to group models within one containing model.

```
MODEL MultiTank

 /* Define one inlet and one outlet port */
 Input1  AS INPUT  FluidStream;
 Output1 AS OUTPUT FluidStream;

 /* Define the Tank sub-models */
 Tank1 AS Tank;
 Tank2 AS Tank;
 Tank3 AS Tank;

 /* Use LINK for external connections outside of  the
containing model */
 LINK Tank1.Input1  AND Input1;
 LINK Tank3.Output1 AND Output1;

 /* Use CONNECT for internal connections within MultiTank
*/
 CONNECT Tank1.Output1 AND Tank2.Input1;
```

```
 CONNECT Tank2.Output1 AND Tank3.Input1;


END
```

You can use LINK to connect to multiports, either where the containing model has a multiport and/or the sub-model has a multiport.

A common situation where you might link to a multiport is where you have a distillation column model that contains an array of stage sub-models. You can define the feed to the column as a multiport. This allows multiple feeds to the stages within the column.

```
MODEL Column

.

Feed AS INPUT MULTIPORT OF Material;

StageMap(Feed.ConnectionSet) AS IntegerParameter(1);

NStages AS IntegerParameter(20);

Stage([1:Nstages]) AS TrayModel;


FOR i IN Feed.ConnectionSet DO

 FeedStage: LINK Feed.Connection(i) AND

            Stage(StageMap(i)).TrayFeedPort;

ENDFOR

.

END
```

In this example, the key is the array StageMap. You can define the stage numbers in StageMap for each of the connections made to the column. For example, if you have three connections to the column's Feed port, you can define which stage these connect to, so that StageMap is an array of three elements indexed by the connections to the Column. For example:

```
StageMap("S1").Value: 5;

StageMap("S2").Value: 11;

StageMap("S3").Value: 14;
```

EXTERNAL Keyword

The External keyword can be used in a definition within a model. The behaviour depends on the type being used for the definition.

If the type is a variable type then the variable definition is a reference to a variable in the parent of this model.

```
e.g. TotalFlow  as EXTERNAL moleflow;
```

See *Using External Properties to Write SubModels*.

If the type is a structure type then the parameter defined can contain a reference to an instance of the structure type.

```
e.g. Rxn1 as External Powerlaw;
```

where Powerlaw is the name of a structure type and Rxn1 is the reference you will be using in this model. Having defined this reference you can use it to access parameters and variables in the structure type.

```
e.g. Rxn1.HeatofReaction
```

where HeatofReaction is a variable or parameter in the structure Powerlaw. This could then be used wherever local parameters or variables are used. However note you cannot assign a value to a parameter or variable in a structure from a reference. You can only do that in the structure type itself.

See *How to use structures* and *Defining Structure Types.*

Connectivity Rules

Connect ports in models as shown in the following table:

To connect ports belonging to	To ports belonging to	Use
Submodels	Submodels	CONNECT
Submodels	Containing model	LINK

Connectivity Rules for CONNECT

When using the CONNECT statement, connect ports and variables according to the following table:

Connect this port / variable	To this port / variable
INPUT	OUTPUT
OUTPUT	INPUT

Connectivity Rules for LINK

When using the LINK statement, connect ports according to the following table:

Link this port	To this port
INPUT	INPUT
OUTPUT	OUTPUT

Using SIGMA

Use the SIGMA keyword in models to sum values in expressions. Typically the expressions contain array variables.

SIGMA Syntax

The syntax for using SIGMA is:

```
SIGMA (Expression1, Expression2, …);
```

Expression ..... An expression

SIGMA Remarks

The SIGMA function is normally applied to expressions containing arrays. You can sum the whole array or a defined slice of an array. If you use an array variable in sigma without any qualification, all the values in the array are summed.

You can apply SIGMA to multidimensional arrays. To control the way SIGMA works, you define the extent of the arrays to be summed. You can also use SIGMA to sum the variables across ports within a multiport.

SIGMA Examples

The most basic use of the SIGMA function is to sum non-array variables or values:

```
A1, A2, A3, B1 AS RealVariable;

B1 = SIGMA (A1, A2, A3, 2.0 );
```

The SIGMA command is interpreted as:

```
B1 = A1 + A2 + A3 + 2.0
```

More typically, SIGMA is used to sum elements of arrays:

```
ComponentList AS StringSet(["N2", "O2", CO2"]);

x(ComponentList) AS Fraction;

SIGMA(x) = 1.0;
```

In this example, x is an array, so SIGMA(x) calculates the sum of the values in the array. The SIGMA command is interpreted as:

```
x("N2") + x("O2") + x("CO2") = 1.0
```

Another way you could write the expression is to list the array elements explicitly:

```
SIGMA(x("N2"),x("O2"),x("CO2")) = 1.0;
```

The following statement is equivalent to the previous SIGMA expression:

```
SIGMA(x(ComponentList)) = 1.0;
```

You can use SIGMA on the whole array, or a selected slice of an array as in the following example:

```
Nsect           AS IntegerParameter(10);
Temp([1:Nsect]) AS Temperature;
X               AS RealVariable;

x = SIGMA(Temp([2:Nsect]));
```

You can remove an element of a set from the calculation in the SIGMA expression:

```
NonWaterFraction AS Fraction;
x(ComponentList) AS Fraction;
SIGMA(x(ComponentList - ["WATER"])) =
  NonWaterFraction;
```

In this example, SIGMA is used to sum the fractions of all the components in a component list except for water.

Using SIGMA for Multidimensional Arrays and Multiports

You can use SIGMA to sum arrays that contain more than one dimension.

```
SIGMA (Variable (Dim1, Dim2, ... ) )
```

or

```
SIGMA( Array1.Array2. ... )
```

Variable	Name of a multi-dimensional array variable
Dimn	Set expression that defines the elements to sum over in the given array dimension
Arrayn.....	An array element in an object path, such as an array of models, an array of ports, a multiport or an array variable

SIGMA for Multidimensional Arrays Examples

You can create an expression that contains more than one dimension that can be expanded:

```
PORT Material
 F AS Flow;
 z(ComponentList) AS Fraction;
END

MODEL Tank
 M, Mc(ComponentList) AS Holdup;
 Total AS RealVariable;
```

```
 Input1 AS INPUT MULTIPORT OF Material;
 Total = SIGMA(Input1.Connection.z);
END
```

In this example, there are two index sets:

- The multiport set Input1.ConnectionSet,which indexes Input1.Connection
- The component set Input1.Connection(i).ComponentList, which indexes Input1.Connection(i).z

In this case SIGMA performs an expansion that generates all possible expansions so that Total would equal the sum of z for all components in all of the connections to the multiport.

The expression is expanded to:

```
Total = Input1.Connection("S1").z("C1")
 + Input1.Connection("S1").z("C2")
 + ...
 + Input1.Connection("S2").z("C1")
 + Input1.Connection("S2").z("C2")
 + ...
```

Another example is where you have a submodel flowsheet in which the submodels have multiports. You get a path:

```
MODEL MultiTank


 Ntank AS IntegerParameter(10);
 Tank([1:Ntank]) AS MultiMixTank;
 CONNECT ...
 Sum1 = SIGMA(Tank.Input1.Connection.z *
Tank.Input1.Connection.Flow)


END
```

In this case, the sigma expression has to evaluate the expression:

```
SIGMA(Tank([1:Ntank]).Input1.Connection(
ConnectionSet).z(ComponentList) *
Tank([1:Ntank]).Input1.Connection(ConnectionSet).Flow)
```

As in the previous example, all combinations of model array, multiport array and component array, are calculated and summed.

Use the ForEach operator to override the default expansion of arrays within SIGMA. For more information, see the ForEach Operator on page 4-103.

Using the SIZE Function

You can use the SIZE function to determine the number of elements in a set. Size can be applied to integer and string sets. It returns an integer value of the number of elements in the set.

Note: Size may not be used in an equation (or procedure call) statement, or in the condition part of a run time If statement.

SIZE Syntax

```
SIZE (SetName)
```

SetName ..... Name of an integer or string set

SIZE Example

In the following example SIZE is used to determine the number of components used in a model, and calculates some default values for the molefractions of the components.

```
MODEL Feed
 x(ComponentList) AS Molefraction;
  :
 x(ComponentList): 1/SIZE(ComponentList);
  :
END
```

The SIZE function calculates the size of the ComponentList Set. A value of one over the number of components is used as a default value for the molefraction array variable.

Assigning Specifications in Models

You can assign a specification to a variable either when you declare it, or after its declaration. Parameters do not have specifications, as their values are always known. You can also specify a value for a variable or parameter when you declare it or afterwards.

Syntax for Assigning Specifications in Models

The syntax for assigning specifications to variables in models is:

```
VariableName.Spec: Specification;
```

or you can shorten this to:

```
VariableName: Specification;
```

The syntax for assigning values to variables or parameters in models is:

```
Name.Value: Value;
```

or you can shorten this to:

```
Name: Value;
```

You can also assign both the value and specification to a variable using the following syntax:

```
VariableName: Value, Specification;
```

VariableName ..... Name of a variable in the model.

Specification ..... Specification for the variable. If you do not specify a value, Free is assumed. Valid values are one of:

Fixed	A variable whose value is known and fixed.
Initial	A variable whose value is known and fixed at time zero for an initialization or dynamic run.
RateInitial	A state variable whose time derivative is known at time zero for an initialization or dynamic run.
Free	A variable whose value is being solved for.

Name ..... The name of a variable or parameter in the model.

Value..... The value assigned to the variable or parameter.

Assigning Specifications in Models Example

The following example shows the declaration of a variable called Temp. Temp is then given a value and a fixed specification. This specification can be overwritten on the block AllVariables form after the model has been instanced as a block.

```
Temp AS Temperature;

Temp:100.0, Fixed;
```

Alternatively the specifications can be supplied as part of the variable declaration.

```
Temp AS Temperature (Value:100.0, Spec:Fixed);
```

Or this can be shortened to:

```
Temp AS Temperature (100.0, Fixed);
```

The specifications in the model need not provide the complete information for the simulation. You may choose to include only the value or the specification, or both in separate statements.

```
Temp AS Temperature;

Temp: 100.0;
```

This means that the temperature is not fixed in the model. You can define Temp as fixed later when you build the flowsheet. If the variable remains not fixed, the value supplied is used as the starting estimate for the solution routines.

```
Temp AS Temperature;

Temp: Fixed ;
```

This means you are making the variable Temp fixed in the model, but you are not assigning it a value. The default value will be used unless you assign the value later in the model, or in the AllVariables table of a block that uses the model.

In the following example, the variable Level is declared and given a value and an Initial specification:

```
Level AS Length (1.5, Initial);
```

The following example shows how the initial value of a derivative is specified using the RateInitial specification:

```
Mass AS Holdup_mass;

Mass.derivative: 0;

Mass.spec: rateinitial;
```

– or –

```
Mass AS Holdup_mass;

$Mass: 0;

Mass : rateinitial;
```

Referring to Sub-Models in a Model Definition

You can refer to variables and parameters in sub-models directly from a model. This is convenient when using submodels to perform common calculations.

Syntax for Referring to Sub-Models in a Model Definition

You then refer to a variable or parameter in a sub-model using the following syntax:

```
SubModelName.VariableName

SubModelName.ParameterName
```

When you declare the sub-model, you can optionally use equalities between sub-model variable names and the containing model variable names. You can also assign values of model parameters to parameters in the sub-model. To do this use the following syntax:

```
SubModelName AS ModelName (SubModelVariable =
ModelVariable, ... , SubModelParameter: ModelParameter, ...
) ;
```

A useful technique is to conditionally instance sub-models dependant upon parameter values. To do this use the following syntax:

```
SubModelName ( [1:(condition)] ) AS ModelName
(SubModelVariable = ModelVariable, ... , SubModelParameter:
ModelParameter, ... );
```

This declares an array of sub-models. The array will be empty when the condition is false (0), because the integer set is then [1:0], which is an empty set. The array will be one element when the condition is true (1), as the integer set is then [1:1]. The declaration will thus instance the sub-model only when the condition is true.

When you want to instance the sub-model only when a parameter X is equal to a value Y, you would use as the condition above the expression (X==Y).

Referring to Sub-Models Examples

The following example shows a sub-model VolCalc being used to calculate vessel volume within the model Tank:

```
MODEL VolCalc


 Vol AS Volume;
 R   AS Length (Fixed);
 h   AS Length (Initial);
 PiValue AS RealParameter;
 VolumeCalc: Vol = PiValue * R^2 * h;


END


MODEL Tank


 k      AS Constant (Fixed);
 height AS Length;
 Radius AS Length;
 PI     AS RealParameter (3.14159);
```

```
// sub-model instance
 Vc AS VolCalc (R=Radius, h=height, PiValue: PI);


 FlowIn  AS INPUT  Fluid;
 FlowOut AS OUTPUT Fluid;
 Vc.vol = FlowIn.flow - FlowOut.flow;
 FlowOut.flow*FlowOut.flow = K*K * Vc.h;

END
```

In the following example, the sub-model used to calculate density is dependent on the parameter Material. If Material is given the value Alkali the model will use the RhoAlkaliCalc submodel, otherwise it will use the RhoAqueousCalc sub-model:

```
MODEL Tank


 Material AS MaterialType (description: "Material
 type used");
 :


//Using sub-model to calculate density
 RhoAl ([1:(Material=="Alkali")]) AS RhoAlkaliCalc
 (RhoAlkali=Rho, X("NaOH”)=Inlet.X("NaOH"));


 RhoAq ([1:(Material<>"Alkali")]) AS RhoAqueousCalc
(RhoAqueous=Rho);
END
```

Using External Properties to Write Sub-Models

Sub-models are often used to package together standard calculations (for example, property or reaction rate calculations).

Normally, a sub-model declares all of the variables it uses locally. To use a sub-model the containing model must then equate its local variables with the variables in the sub-model. To reduce the duplication of variables and improve simulation efficiency, you can use the EXTERNAL declaration qualifier in the sub-model. The EXTERNAL qualifier indicates that variables do not exist in the sub-model but instead come from the parent of this model.

Note: When you declare the variables of the sub-model as EXTERNAL, you must not include them in equivalence equations in the containing model.

Example of Using External Properties to Write Sub-Models

Normally, a sub-model declares all of the variables it uses locally, for example:

```
Model FractionToFlow

  TotalFlow           as flow_mol;

  Flow(componentlist) as flow_mol;

  X (componentlist)   as fraction;

  Flow = TotalFlow * X;

End
```

To use this sub-model, the containing model must equate its local variables with the variables in the sub-model, for example:

```
Model Simple

 TotalFlow           as flow_mol;

 Flow(componentlist) as flow_mol;

 X(componentlist)    as fraction;


 convert as FractionToFlow (TotalFlow=TotalFlow,
 X=X, Flow=Flow);

   :

End
```

This has introduced duplicates of each Flow and X variable and the TotalFlow variable. To reduce the duplication of variables, you can use the EXTERNAL declaration qualifier in the sub-model, for example:

```
Model FractionToFlow

 TotalFlow           as EXTERNAL moleflow;

 Flow(componentlist) as EXTERNAL flow;

 X (componentlist)   as EXTERNAL fraction;

 Flow = TotalFlow * X;

End
```

The EXTERNAL qualifier indicates that the variables TotalFlow, Flow, and X do not exist in FractionToFlow but instead come from the parent of this model, in this case the model Simple. For example, where TotalFlow is used in FractionToFlow, it is TotalFlow in Simple that is used.

When you declare the properties of the sub-model as EXTERNAL, you must remove the equivalence equations in the containing model:

```
Model Simple
 TotalFlow           as moleflow;
 Flow(componentlist) as flow;
 X(componentlist)    as fraction;

 convert as FractionToFlow;
  :
End
```

Now there is only one copy of the variables TotalFlow, Flow, and X.

Changing the Default Mapping For External Properties

The containing model can redefine a sub-model's external properties using the IS operator in the declaration of the sub-model.

Example of Changing the Default Mapping for External Properties

The following example demonstrates extending the example given in Using External Properties to Write Sub-Models on page 4-121 so that the sub-model can be used twice in the same model:

```
Model Simple

 TotalX              as flow_mol;
 TotalY              as flow_mol;
 FlowX(componentlist) as flow_mol;
 FlowY(componentlist) as flow_mol;
 X(componentlist)    as fraction;
 Y(componentlist)    as fraction;

 convertX as FractionToFlow (TotalFlow IS TotalX,
 Flow IS FlowX);
 convertY as FractionToFlow (TotalFlow is TotalY,
 Flow IS FlowY, X IS Y);
  :
End
```

Here convertX.TotalFlow will map to TotalX and convertY.TotalFlow will map to TotalY.

Notes and Restrictions on Writing Sub-Models

Note the following points when writing sub-models:

- An assignment to an external property within a sub-model may conflict with other assignments made to the same property elsewhere. Although you can assign external properties in the sub-model, you must be careful. The sub-model assignment can conflict with an assignment to the same external property from another sub-model. If assignments conflict no warning is given and the result may be unpredictable.
- External arrays must be declared with consistent indexing in the model and sub-model. If the indexes of the external property and the resolved property do not match then array operations may be expanded incorrectly. The parser does not check the indexes for consistent use: this is the responsibility of the model writer.
- When you edit models, if you add or remove external properties, you *must* recompile any models that use the edited model.
  - Recompiling the models ensures that any changes are completely propagated.
- Models that use external variables can only be used as sub-models: they cannot be placed directly onto the flowsheet.

Using Virtual Types in Sub-Models

Virtual types are used in sub-models where the types of the properties can only be determined when the model is included in a containing model.

Example of Using Virtual Types in Sub-Models

The following model is an example usage of VIRTUAL types. The sub-model calculates the average of a vector of values:

```
Model AverageValue

 VIRTUAL ValueType USES RealVariable;

 NValues             as external IntegerParameter;

 Value([1:NValues]) as external RealVariable;

 Average             as ValueType;

 Average = sigma(Value)/NValues;

End
```

The first statement declares that the type ValueType is a VIRTUAL type derived from RealVariable. Properties of type ValueType (that is, Average) are referred to as virtual properties. By using a virtual type, it is then possible for the containing model to define the types that will be used for virtual properties.

For example, the following model uses the AverageValue sub-model to calculate the average temperature from a temperature profile:

```
Model Pipe

 // Discretize temp profile into N sections
 NSections        as integerParameter;
 T([1:NSections]) as Temperature;
 Q                as heat_flux;
 WallArea         as Area;
 U                as notype;
 T_Wall           as Temperature;

 // Use sub-model to calculate average
 T_Av as AverageValue (NValues is NSections,
                       Value is T);
 // Calculate heat transfer based on average temp
 Q = U * WallArea * (T_Wall - T_Av.Average);

End
```

Since ValueType has not been redefined, T_Av.Average is a simple RealVariable. This means that it will appear on tables and plots with no units of measurement. The ValueType can be redefined by using the IS operation in the declaration of the sub-model, for example:

```
Model Pipe

 // Discretize temp profile into N sections
 NSections          as integerParameter;
 T([1:NSections])   as Temperature;
 Q                  as heat_flux;
 WallArea           as Area;
 U                  as notype;
 T_Wall             as Temperature;
```

```
// Use sub-model to calculate average
T_Av as AverageValue (NValues IS NSections,
Value IS T, ValueType IS Temperature);


// Calculate heat transfer based on average temp
Q = U * WallArea * (T_Wall - T_Av.Average);


End
```

Here ValueType has been redefined to be Temperature (which is derived from RealVariable). Consequently, T_Av.Average is created as a Temperature variable rather than as a simple RealVariable, so it will show temperature units of measurement on tables and plots.

Restrictions on Using Virtual Types

Note the following restrictions when using virtual types:

- **Virtual types are only visible in the scope in which they are declared**: they cannot be used directly within nested model definitions. However, the right side of the IS operator may be a virtual type. In this way, the redefinition of a virtual type in a container model can be propagated down to sub-models nested within sub-models.
- When you edit models, if you add or remove virtual properties, you must recompile any models that use the edited model.
  - Recompiling the models ensures that any changes are completely propagated through the system.

Using SWITCH

Use switch to define different sets of equations that will be used dependant upon the state of your model. You define an initial state, and the conditions under which the state will change.

Switch is active in all run modes, but will probably be most useful in dynamic simulations.

Syntax for SWITCH

```
SwitchName: SWITCH


  INITIAL STATE StateName
   SwitchStatements;
    IF Condition1 STATE: NewState1;
    IF Condition2 STATE: NewState2;
     :
  ENDSTATE
```

```
  STATE StateName
   SwitchStatements;
    IF Condition3 STATE: NewState3;
    IF Condition4 STATE: NewState4;
     :
  ENDSTATE

   :

ENDSWITCH
```

Switchname	Optional name you can give to the Switch definition.
INITIAL	The state that applies at the start of the simulation run. Use the Initial keyword on only one state definition. **Note:** One of the states you define in a SWITCH definition must be flagged as the initial state.
StateName	Unique name of a state within the current Switch definition.
SwitchStatements	Equations that apply when in the state you are defining.
Condition	Condition for switching from the current state to the new state.
NewState	Name of the new state to switch to if the condition is met.

SWITCH Remarks

SWITCH provides a quick and easy way for you to describe both reversible and non-reversible changes in your process.

Note that you cannot change the structure of the simulation between states. Each state definition in a switch must maintain a square and complete simulation status. This means that you must have the same number of equations in each state within a SWITCH definition.

SWITCH Examples

The following example shows how to model a non-reversible action using a switch definition. The switch is non-reversible because there is a condition that enables the state Normal to switch to Burst, but there is no condition to reverse this switch.

```
SWITCH
 INITIAL STATE Normal
  VentFlow = 0;
  IF VesselPres > 8.5 STATE: Burst;
 ENDSTATE
```

```
 STATE Burst
  VentFlow = 0.01*(VesselPres - 1.0)/8.0;
 ENDSTATE
ENDSWITCH
```

The following example shows a switch statement used to model a tank with two different cross-sectional areas. Between fluid height 0.0 and 1.0, the sectional area is 0.25; between the fluid level 1.0 and 11.0, the sectional area is 2.0. The switch models overflow in this example, where the fluid level remains constant.

```
CalculateFluidLevel:
SWITCH
 INITIAL STATE BottomZone
  FluidLevel = FluidVolume/0.25;
  OverflowFlowrate = 0.0;
  IF (FluidVolume >= 1.0) STATE: Topzone;
 ENDSTATE
 STATE TopZone
  FluidLevel = (FluidVolume - 1)/2 + 4;
  OverflowFlowrate = 0.0;
  IF FluidVolume >= 11.0 STATE: Overflowing;
  IF FluidVolume < 1.0   STATE: BottomZone;
 ENDSTATE
 STATE Overflowing
  OverflowFlowrate = MAX(FluidInA.Flow +
  FluidInB.Flow - FluidOut.Flow,0);
  FluidLevel = 9.0;
  IF FluidVolume < 11.0 STATE: TopZone;
 ENDSTATE
ENDSWITCH
```

5 Modeling PDE Systems

Overview of PDE Modeling

Aspen Modeler products provide a systematic way to model and solve distributed parameter systems in which process variables vary with respect to spatial position as well as time. Such systems are common in engineering and science and are described mathematically by partial differential equations (PDEs).

In process engineering, PDEs are the fundamental basis for describing complex physical phenomena such as mass, heat, and momentum transport coupled to chemical reactions in 1D, 2D, and 3D. Process equipment items that you can model in detail using PDEs include tubular reactors, packed bed reactors, adsorption columns, fuel cells, crystallizers, heat exchangers, and pipelines.

With the custom PDE modeling feature in Aspen Modeler products, you can formulate coupled linear and nonlinear partial differential equations for use in steady-state and dynamic simulations. The PDE modeling language enables you to write compact and readable PDEs over one, two, or three spatial dimensions, using rectangular or polar coordinates. The built-in text editor, configuration forms, and specification analysis also facilitate the efficient preparation of your PDE models for solution.

This chapter describes:

- How PDE modeling is implemented.
- Creating a custom PDE model.
- Using domains and distributions with PDE modeling.
- Integral Partial Differential Equation (IPDAE) models.
- Using Method of Lines for the solution of PDEs.

About PDE Modeling

Aspen Modeler products offer a number of powerful features for modeling, solving and analyzing PDE models. Domains and distributions are built-in sub-models provided to make it quick and easy to build your own PDE models. Domains determine how to discretize spatial dimensions based on your selection of the type and order of discretization method, as well as element spacing. Distributions represent variables distributed over a single domain or

up to three domains in the case of a three-dimensional distribution. They contain the logic for calculating partial derivatives with respect to the independent spatial variables.

The well-known Method of Lines (MOL) is used to solve the time-dependent PDE systems arising from distributed parameter models. It consists of the following two-phase solution method:

- In the first phase, the partial differential equations in your models are discretized automatically using the finite difference or finite element method you select.
- In the second phase, the resulting DAE systems are integrated over time using the dynamic solver of your choice.

The discretization methods and dynamic solvers can be selected and controlled in order to assure convergence for different types of problems.

Profile plotting is available for visually examining the results of PDE simulations. You can use 2D color profile plots to view the distribution of your process variables along the length of a discretized domain. With 3D color profile plots you can analyze a PDE solution over two spatial dimensions or over a single spatial dimension and time.

Creating a Custom PDE Model

To create a new custom PDE model:

1 Create a model.

2 Declare one or more domains. To customize a domain, you can specify the following properties:

- Highest order derivative
- Discretization method
- Domain length
- Number and location of domain section(s)
- Element spacing preferences

3 Declare a distribution for each distributed variable. To customize a distribution, you can specify the following properties:

- Highest order derivative of the variable in each direction if it differs from the default set for the corresponding domain
- Integral calculations

4 Write your partial differential equations over the domain(s).

5 Write your boundary conditions.

6 For a dynamic simulation, write your initial conditions.

Example of Custom PDE Modeling

The following example shows a complete description of a two-dimensional heated slab:

```
MODEL HeatedSlab
```

```
// Declare domains for X and Y spatial domains
X as LengthDomain (DiscretizationMethod:"BFD1",
 HighestOrderDerivative: 2, Length:1,
SpacingPreference:0.1);
Y as LengthDomain (DiscretizationMethod:"BFD1",
 HighestOrderDerivative: 2, Length:1,
SpacingPreference:0.1);

// Declare a two-dimensional distribution for temperature
T as Distribution2D (XDomain is X, YDomain is Y) of
  Temperature(10);
D as constant (1.2, Fixed);

// Heat transfer equation: Time-dependent second order
partial
// differential equation. PDE is specified over the
interior of
// the domain.
$T(X.Interior,Y.Interior) = D * T(X.Interior,
Y.Interior).d2dx2
                          + D * T(X.Interior,
Y.Interior).d2dy2;

// Initial conditions - steady state
T(X.Interior, Y.Interior): RateInitial;

// Dirichlet boundary conditions
// Specify temperature at domain boundaries
T(0,         [0:Y.EndNode]): Fixed, 31;
T(X.EndNode, [0:Y.EndNode]): Fixed, 15;

// Neumann Boundary conditions - No heat transfer at sides
T([1:X.EndNode-1], 0).ddy         = 0;
T([1:X.EndNode-1], Y.EndNode).ddy = 0;

End
```

Using Domains with PDE Modeling

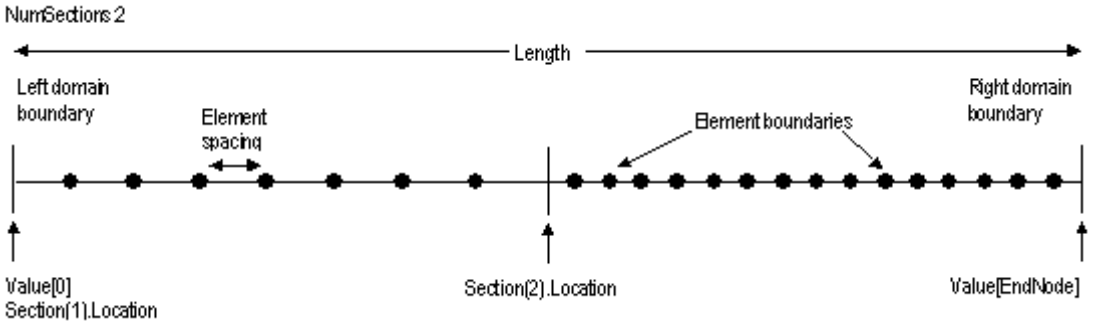
Domains are used to define the space over which a PDE system and its corresponding variables are distributed. A single domain describes one dimension of the distribution (for example, the axis of a tubular reactor). In the modeling language, a built-in type definition called Domain represents this concept. Two other built-in types, LengthDomain and AngleDomain, inherit properties from Domain and are provided for your modeling convenience.

For example, a pipe model that contains equations that are distributed along the axis may contain the following declaration:

```
Model Pipe
 Axis as LengthDomain (Length:10,
  SpacingPreference:1.0, NumSections:1,
  DiscretizationMethod:"CFD4");
End
```

In this example, Axis is an instance of the LengthDomain model with boundaries at 0.0 (the default) and 10.0. Describing the boundaries of a domain is only a small part of the functionality of a domain model. It also calculates how the space is discretized. The discretization is based on your selection of element spacing, number of domain sections, and type and order of discretization method.

The following is an illustration of using a domain with two sections. The element spacing for the second section is half that of the first section.



Notes:

- You can change domain properties in the domain declaration or by using the domain configuration form.
- The built-in Domain model is dimensionless. The units of measurement for the LengthDomain and AngleDomain are meters and radians, respectively. You must ensure that these base units of measurement are consistent for all your model

equations. The user interface converts between the base units of measurement and your current UOM so that plots and tables are displayed in the chosen UOM.

See *Using Units of Measurement in Chapter 2* for more information.

Declaring Domains for PDE Modeling

When modeling a distributed parameter system, you start by declaring one or more domains. For each domain, you can specify the following properties:

- Discretization method.
- Highest order derivative.
- Domain length.
- Element spacing preference.
- Number of sections.
- Section location and spacing.

The ability to assign these properties separately for each domain instance gives you considerable modeling flexibility.

You can declare domains in a model using the following syntax.

Domain Syntax

```
Domain, Domain, … AS
              DomainModel OF RealParameterType (Property:
PropertyValue, ... );
```

Domain ..... Name you give to this domain.

DomainModel ..... Name of a defined domain model. Three domain models are built in to the system, namely DOMAIN, LENGTHDOMAIN, and ANGLEDOMAIN. The Domain model is dimensionless. The units of measurement for the LengthDomain and AngleDomain models are meters and radians, respectively.

***RealParameterType*.....** Name RealParameter type to use for domain values and length.

***Property*.....** Parameter or variable defined in the built-in domain model. The following are domain parameters that can be used in the assignment list:

DiscretizationMethod	Discretization method used to calculate partial derivatives and integrals. The default is "BFD1". Valid values are "BFD1", "BFD2", "CFD2, "CFD4", "FFD1", "FFD2", "OCFE2", "OCFE3", "OCFE4", and "UBFD4".
HighestOrderDerivative	Highest-order partial derivative (with respect to the independent spatial variable) to calculate for a given domain. Valid values are 0, 1 and 2. The default value is 1. No partial

		derivatives are calculated for a value of 0. For a value of 1, only 1st-order partial derivatives are calculated. A value of 2 means that both 1st-order and 2nd-order partial derivatives are calculated.
	Length	Length of domain. The default is 1.0.
	NumSections	Number of sections in the domain. Default value is 1.
	SpacingPreference	Preferred spacing for the domain. The default value is Length/NumSections/8. Note that the actual spacing used (SpacingUsed) by Aspen Custom Modeler may differ from your preferred spacing. This happens when your domain length is not a multiple of your spacing preference, and it also depends on the order of the discretization method. The spacing is chosen so there is an integer number of element groups, where a group is defined as the same number of elements as the discretization order. Your preferred spacing must also provide the minimum number of elements required for the discretization method (for example, four elements for 4th-order methods). If it does not, the spacing used is equal to the domain length divided by the minimum number of elements. Because the value of the SpacingUsed property is determined by the built-in domain model, you must not modify it. For more information, see Specifying Element Spacing Preference.
	Section(*).Location	Absolute location of the left-hand boundary of a section. The locations of contiguous sections in a given domain must be in increasing order. The location of the right-hand domain boundary is equal to Section(1).Location plus Length.
	Section(*).Spacing Preference	Preferred spacing for the section. The default value is the value of the SpacingPreference property for the domain. Element spacing is uniform in any given section. Different spacing can be used in different sections. Note that the actual spacing used, Section(*).SpacingUsed, may differ from your preferred section spacing for the same reasons discussed for the SpacingPreference property. Because the value of the Section(*).SpacingUsed property is determined by the built-in domain model, you must not modify it. For more information, see Element Spacing Preference
PropertyValue	Value to be assigned to the property**.**	

Domain Remarks

- You can declare a comma-separated list of domains of the same type in a single declaration statement.
- You may associate up to three domains with each distributed variable in your process model. This association is made with a distribution.
- Distributed variables can refer only to domains that you have already declared explicitly in your model.
- You can change the configuration of a domain using the domain configuration form. This form is particularly useful if you want to use different discretization methods or element spacing.

Domain Declaration Examples

All the following declarations appear within a model definition.

In the first example, a domain called X is created in the model. X takes on the characteristics of the built-in definition of a domain model:

```
X AS Domain;
```

In this example, the domain properties such as X.Value and X.Length are simple RealParameters. To provide units of measurement for a Domain the value type can be redefined using the 'of' qualifier, for example:

```
X AS Domain of LengthParameter;
```

For brevity, LengthDomain is provided as a built-in DomainModel that uses LengthParameter as its ValueType by default. The following example declares Axis as a domain of type LengthDomain, and requires 2nd-order partial derivatives:

```
Axis AS LengthDomain (HighestOrderDerivative:2);
```

In the next example, the domain Angle is of type AngleDomain, which is a built-in domain model. The default discretization method used is the 4th-order central finite difference method.

```
Angle AS AngleDomain(DiscretizationMethod:"CFD4");
```

The following example shows three domains defined in one statement. Each domain contains 2nd-order partial derivatives that are discretized using 4th-order orthogonal collocation on finite elements:

```
X, Y, Z AS LengthDomain (HighestOrderDerivative:2,
DiscretizationMethod:"OCFE4");
```

By using three separate domain declaration statements, a different discretization method can be used for each domain:

```
X AS LengthDomain (HighestOrderDerivative:2,
DiscretizationMethod:"BFD1");

Y AS LengthDomain (HighestOrderDerivative:2,
DiscretizationMethod:"CFD4");

Z AS LengthDomain (HighestOrderDerivative:2,
DiscretizationMethod:"UBFD4");
```

In the next example, the domain Y of type LengthDomain is discretized using the 4th-order upwind biased finite difference method. The values of highest

order derivative and domain length are specified. The domain is divided into three sections having lengths of 3.0, 1.0, and 6.0, respectively. Note here that by default Section(1).Location is 0.0. A domain spacing preference of 0.2 is applied to the first and third sections, whereas a finer element spacing of 0.1 is preferred for the second section.

```
Y AS LengthDomain (DiscretizationMethod:"UBFD4",
HighestOrderDerivative:2, Length:10.0, NumSections:3,
SpacingPreference:0.2, Section(2).Location:3.0,
Section(3).Location:4.0, Section(2).SpacingPreference:0.1)
```

Using Discretization Methods for PDE Modeling

Aspen Modeler products provide two well-known classes of discretization methods for approximating partial derivatives and integrals: finite differences over uniform grids and orthogonal collocation on finite elements (OCFE). As shown in the following table, the methods differ in the order and type of approximation used. For the built-in domain models, the default method is 1st-order backward finite difference (BFD1). You can specify another discretization method when declaring your domain or by going to the domain configuration form.

Method	Order of Approximation	Type of Approximation
BFD1(default)	1st-order	Backward Finite Difference
BFD2	2nd-order	Backward Finite Difference
CFD2	2nd-order	Central Finite Difference
CFD4	4th-order	Central Finite Difference
FFD1	1st-order	Forward Finite Difference
FFD2	2nd-order	Forward Finite Difference
UBFD4	4th-order	Upwind Biased Finite Difference
OCFE2	2nd-order	Orthogonal Collocation on Finite Elements
OCFE3	3rd-order	Orthogonal Collocation on Finite Elements
OCFE4	4th-order	Orthogonal Collocation on Finite Elements

Discretization Method Remarks

- The order of approximation for partial derivatives in finite difference methods and the degree of polynomials used in finite element methods have a significant impact on the accuracy of the solution. Generally, a higher order algorithm of the same type implies higher accuracy. This is especially true if you use a large element spacing (coarse grid).
- The discretization method specified in the domain for calculating partial derivatives is also used to compute the integral over each element in any distribution using that domain.

- By writing your PDEs and initial conditions using the Interior set provided in the built-in Domain model, you can switch between the various discretization methods without having to modify your model. Aspen Modeler products automatically handle the underlying requirement that finite difference and OCFE methods inherently require derivatives, integrals and model equations to be evaluated at different discretization points.
- You can use different discretization methods for each domain in a 2D or 3D distribution.

Specifying a Discretization Method in the Domain Configuration Form

Use the domain configuration form to choose a PDE discretization method for your domain:

1. In the All Items pane of the Simulation Explorer, double-click on the Flowsheet folder.
2. In the Contents pane, double-click Configure for the domain block of interest. The domain configuration form appears.
3. Click the cell corresponding to the DomainName.DiscretizationMethod (row) and the Value attribute (column).
4. Select the method you want from the menu.

Examples of Specifying a Discretization Method in the Declaration of a Domain

When writing a PDE model, the discretization method can be specified in the declaration of the domain.

In the first example, a domain called X is discretized using the 4th-order central finite difference method. CFD4 is used to calculate partial derivatives and integrals (if integral calculations are turned on for any distributions using domain X).

```
X AS Domain(DiscretizationMethod:"CFD4");
```

The following example shows two domains declared in one statement. Each domain contains 2nd-order partial derivatives that are discretized using 2nd-order orthogonal collocation on finite elements:

```
Axial, Radial AS LengthDomain (HighestOrderDerivative:2,
DiscretizationMethod:"OCFE2");
```

By using two separate domain declaration statements, a different discretization method can be used for each domain:

```
Axial AS LengthDomain (HighestOrderDerivative:2,
DiscretizationMethod:"BFD1");

Radial AS LengthDomain (HighestOrderDerivative:2,
DiscretizationMethod:"CFD4");
```

Specifying Domain Length

Distributed parameter systems are described as distributions over one or more spatial domains of a given length.

Taking a tubular reactor as an example, the lengths of the axial and radial domains are the reactor length and radius, respectively.

The built-in domain models contain a property called Length. The default value for the domain length is 1.0. This value can be changed in the declaration of your domain or by using the domain configuration form. **Tip:** While the built-in Domain model is dimensionless, the units of measurement for the LengthDomain and AngleDomain models are meters and radians, respectively. When setting the Length parameter for the built-in LengthDomain and AngleDomain models, you must ensure that these base units of measurement are consistent for all your model equations. The user interface converts between the base units of measurement and your current UOM so that plots and tables are displayed in the chosen UOM. See Using Units of Measurement in Chapter 2 for more information.

Specifying Domain Length in the Domain Configuration Form

Use the domain configuration form to specify the length of your domain. The default value is 1.0.

1. In the All Items pane of the Simulation Explorer, double-click on the Flowsheet folder.
2. In the Contents pane, double-click Configure for the domain block of interest. The domain configuration form appears.
3. Click the cell corresponding to DomainName.Length (row) and the Value attribute (column).
4. Enter the domain length you want.

Examples of Specifying Domain Length in the Declaration of a Domain

When writing a PDE model, the length of the discretized domain can be specified in the declaration of the domain. If it is not specified, the length defaults to the value of 1.0 assigned in the built-in domain models.

In the first example, a domain X of length 25 is discretized using the 4th-order central finite difference method.

```
X AS Domain(DiscretizationMethod:"CFD4", Length: 25.0);
```

The next example shows the specification of the domain lengths for a two-dimensional reactor problem.

```
Axial AS LengthDomain (Length:10.4);

Radial AS LengthDomain (Length:2.1);
```

You can also specify the domain length as a parameter:

```
R as lengthparameter;

Radial AS LengthDomain (Length:R);
```

Specifying Element Spacing Preference

Given a fixed domain length, the spacing of the elements determines the granularity of the discretized grid and the number of elements. A small spacing gives a fine grid with a large number of elements, whereas a larger spacing means a coarse grid with a small number of elements. The element spacing, along with the domain length, number of sections, and section locations, determine the number and location of discretization nodes.

Element spacing is uniform within a given domain section, but non-uniform spacing can be applied using multiple sections, each with a different spacing. The latter is particularly useful for modeling PDE systems with sharp fronts.

The element spacing you use can have a large influence on the trajectory of the solution. Generally, a smaller element spacing implies higher accuracy at the expense of additional computation. As you reduce the element spacing, the computational requirements increase. This happens because the number of partial derivatives to be computed is in general proportional to the number of elements. On the other hand, if you specify too large a value, the simulation may be far from accurate and even fail to converge, especially for problems with steep gradients.

Aspen Modeler products enable you to specify an element spacing preference for your domain. The default value of the SpacingPreference property is Length/NumSections/8, a spacing that gives 8 points per section. A different value can be specified in the declaration of your domain or by using the domain configuration form. Different element spacings can be used in different sections, but element spacing is uniform in any given section.

Note: The actual spacing used (SpacingUsed) may differ from your preferred spacing. This happens when your domain length is not a multiple of your spacing preference times the method order. In other words, the spacing will be chosen so that the number of elements is an integer for a first order method, an even integer for a second order method or an integer multiple of 4 for a fourth order method.
In this case, the domain length is divided by your preferred spacing multiplied by the method order. The result is rounded to the nearest integer to determine the number of element groups of size 1 (first order), 2 (second order), 3 (third order) or 4 (fourth order). The domain length is then divided by the calculated number of elements to produce the actual spacing used.
If the discretization method is OCFE or order 3 or higher, this is the average spacing, as the points are non-equally spaced within each group of 3 or 4.
Your spacing preference must also give enough elements to support the discretization method you select. If it gives fewer, the actual spacing used is set to give the minimum. For example, all of the 4th-order discretization methods require at least four elements.
In summary, the value of the SpacingUsed property depends not only on the preferred spacing you specify, but also on domain

length, as well as the order and type of discretization method you select. Because the value of the SpacingUsed property is determined by the built-in domain model, you must not modify it.

Specifying Spacing Preference in the Domain Configuration Form

Use the domain configuration form to specify the element spacing preference. The default value is Length/NumSections/8.

1. In the All Items pane of the Simulation Explorer, double-click on the Flowsheet folder.
2. In the Contents pane, double-click Configure for the domain block of interest. The domain configuration form appears.
3. Click the cell corresponding to the DomainName.SpacingPreference (row) and the Value attribute (column).
4. Enter the element spacing you prefer to be used for this domain.

Examples of Specifying Spacing Preference in the Declaration of a Domain

When declaring a domain in your PDE model, you can specify a preferred element spacing in the list of property assignments.

In the first example, the preferred element spacing (SpacingPreference) for domain X is set to a value of 0.2. Because the length is a multiple of the preferred element spacing, the actual spacing used (SpacingUsed) is also 0.2. The total number of elements in this domain is equal to Length/SpacingUsed or 2.0/0.2 = 10.

```
X AS Domain(DiscretizationMethod:"BFD1", Length:2.0,
SpacingPreference:0.2);
```

For the higher order discretization methods, the elements are considered to be arranged into groups of 2,3 or 4 depending on the method order. If the length is not a multiple of the spacing preference you choose multiplied by the method order, the SpacingUsed property is based on the nearest number of element groups.

For example, if your domain length is 1.0 and you specify a preferred spacing of .35 for BFD1, the actual SpacingUsed property will be set to .3333 (1.0/3). If your domain length is 1.0 and you specify a preferred spacing of 0.0875 and the discretization method is CFD4, then the SpacingUsed property will be set to 0.08333 (`1.0/12), giving three groups of four elements. If the discretization method is OCFE4, this is the average spacing, but the spacings within each group of four elements will vary according to the position of the collocation points.

As another example, take the case where your domain length is 1.0 and you specify a spacing preference of 0.5 with a 4th-order discretization method. The value of the SpacingUsed property in this case is set to .25 (1.0/4), because at least four discretization points are required for a 4th-order method.

```
Axial AS LengthDomain (DiscretizationMethod:"CFD4",
Length:1.0, SpacingPreference:0.5);
```

The next example shows the domain length and spacing specified as parameters:

```
R as lengthparameter;

n as integerparameter(10);

Radial AS LengthDomain (Length:R, SpacingPreference:R/n);
```

Using Domain Sections

The solution path for some PDEs may exhibit steep gradients in one or more spatial domains. When the location of a sharp front does not move much in time and is known a priori, you can divide the domain into multiple sections to handle this problem. You can use a small element spacing (that is, fine grid) for any section that contains a steep gradient and a large spacing (that is, coarse grid) elsewhere.

The built-in Domain model contains the NumSections property for specifying the number of sections. The default value is 1. You have the flexibility to set the Location and preferred element spacing (Section(*).SpacingPreference) for each section in your domain. The default location for the first section (that is, Section(1).Location) is 0.0. By default, the remaining section locations are set at equally spaced intervals. The default spacing preference for each section is taken to be the spacing preference specified for the domain.

Section Remarks

- You can change the number of domain sections and their location and element spacing in the declaration of your domain or by using the domain configuration form.
- The locations of contiguous sections in a given domain must be in increasing order. A section location value must not exceed Section(1).Location plus Length.
- Uniform element spacing is used in a given section. Element spacing can vary between sections, therefore, non-uniform spacing can be applied using multiple sections, each with a different spacing. The latter is particularly useful for modeling PDE systems with sharp fronts.

Specifying Section Parameters in the Domain Configuration Form

Use the domain configuration form to specify the number of sections and their location and element spacing.

1 In the All Items pane of the Simulation Explorer, double-click the Flowsheet folder.

2 In the Contents pane, double-click Configure for the domain block of interest. The domain configuration form appears.

3 Click the cell corresponding to the Value attribute (column) of the DomainName.NumSections (row).

4 Enter the number of sections you want.

5 For each of the domains, click the cells corresponding the Value attribute (column) of the DomainName.Section(*).Location (row) and enter the location. The value you enter represents the absolute location of the left-hand boundary of the section. The locations of contiguous sections in a given domain must be in increasing order and a section location value must not exceed Section(1).Location plus Length.

6 For each of the domains, click the cells corresponding the Value attribute (column) of the DomainName.Section(*).SpacingPreference (row) and enter the element spacing you prefer.

Examples of Specifying Section Parameters in the Declaration of a Domain

When declaring a domain with multiple sections, you can specify the number of sections using the NumSections parameter in the declaration of the domain.

The first example shows that domain X is divided into three sections.

```
X AS Domain(NumSections:3);
```

You can set the location of one or more of the sections in the domain using the parameters called Section(*).Location. In the following example, the second and third sections are located at 0.3 and 0.5, respectively. The default location for Section(1) is 0.0.

```
X AS LengthDomain(NumSections:3, Section(2).Location:0.3,
Section(3).Location:0.5);
```

Next, a different spacing preference is specified for each of the three sections in the previous example. A much smaller spacing between elements is used for the second section because this part of the domain is known to contain a sharp front. Note here also that the Section(*).Location values must not exceed 1.0 as the domain Length is 1.0 by default.

```
X AS LengthDomain(NumSections:3, Section(2).Location:0.3,
Section(3).Location:0.5, Section(1).SpacingPreference:0.2,
Section(2).SpacingPreference:0.01,
Section(3).SpacingPreference:0.2);
```

In the following example, two sections each having size 8.0 are specified. The spacing preference for the first section is set at a value of 0.2, whereas the spacing preference for the second section is 1.0 by default (Length/NumSections/8).

```
X AS LengthDomain(DiscretizationMethod:"CFD4", Length:16.0,
NumSections:2, Section(1).Location:-8.0,
Section(2).Location:0.0, Section(1).SpacingPreference:0.2)
```

Number and Location of Discretization Nodes

Based on the configuration parameters you select, the Domain model distributes the discretization nodes. More specifically, it determines the number of nodes (EndNode+1) and their location, Value([0:EndNode]), where

0 and EndNode correspond to the node indices for the left-hand and right-hand domain boundaries, respectively. Given this convention for indexing over the discrete domain, Value([0:EndNode]) specifies a closed domain and Value([1:EndNode-1]) defines a domain open on both ends. The Value array and EndNode are displayed in the Configuration Form for each domain.

EndNode and Value Array Remarks

- The EndNode parameter and the Value array are calculated by the domain and must not be modified.
- If you wish to produce a profile plot showing your distribution, use the Value array in your domain as your X-axis variable. For example, use XDomain.Value ([0:XDomain.EndNode]) or XDomain ([0:XDomain.EndNode]) for short.

Example of Use of EndNode and Value Array

The following example of Dirichlet boundary conditions demonstrates the use of EndNode and the Value array:

```
// Boundary conditions specified using EndNode

T(0,[0:Y.EndNode] ) = 0.0;

T([1:X.EndNode-1],0) = 0.0;

T([1:X.EndNode-1],Y.EndNode) = 0.0;

// Use the Value array, Y([0:Y.EndNode]), to vary the
boundary condition depending

// on location along the boundary

T(X.EndNode, [0:Y.EndNode]) =
sinh(PI)*sin(PI*Y([0:Y.EndNode]));
```

Aspen Custom Modeler Reference

→Modeling Language Reference

→Modeling PDE Systems

→Using Domain with PDE Modeling

Interpolation of Existing PDE Models

When building PDE models, it is desirable to examine the effect of changing the mesh spacing preference used. This is necessary when trying to determine the optimum number of nodes required whilst maintaining numerical dispersion at an acceptable level.

Should the model prove difficult to converge, when changing the mesh spacing preference used it is desirable to have good initial guesses for any new nodes introduced.

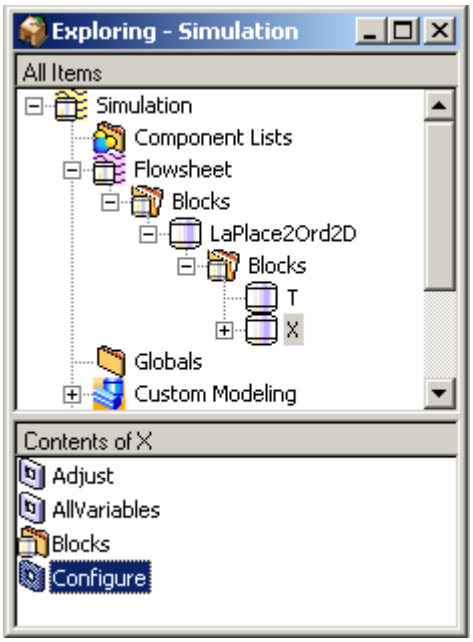
Within the domain model, a method has been provided to allow you to interpolate the new mesh from a previously solved mesh.

How to Interpolate a PDE Problem

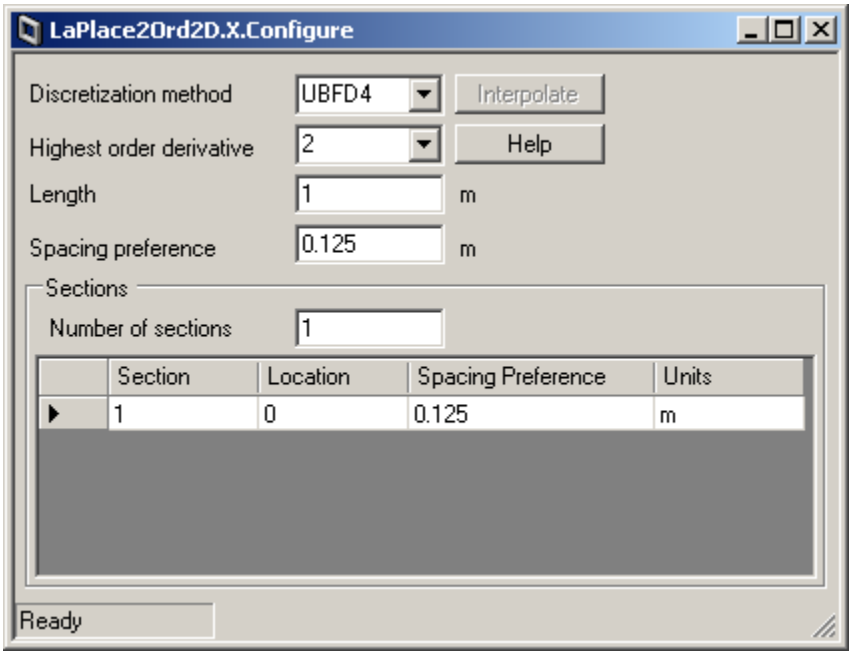
To modify and interpolate an existing domain:

1 Explore the model and locate the domain to be modified.

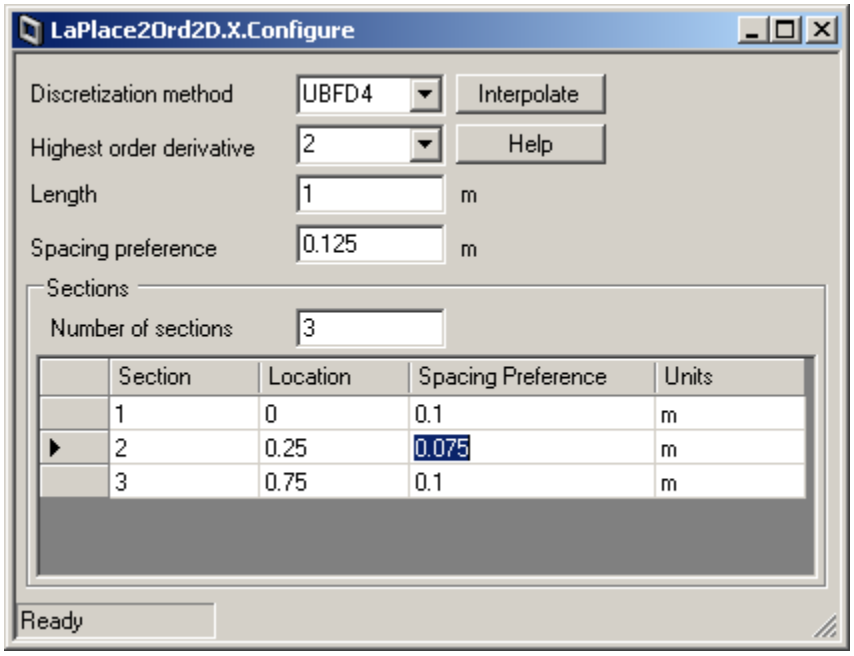
2 Open the form named Configure:



3 Make required changes to the domain using the Configure form:



4 When changes are complete press the now active "Interpolate" button:



5 Close the configure form.

What is Supported?

The domain interpolation will support dependent distributions and variable arrays of 1 to 3 dependent dimensions.

Dependent distributions and variable arrays at the same model level as the domain and lower will only found and interpolated. If the simulation message window print level is medium or higher, messages indicating the dependents found will be displayed.

When creating variable arrays to represent distributed variables the array dimensions must be explicitly declare using the either the Interior set or EndNode parameter of the domain. For example:

```
//Declare domain
x As Domain;
//Valid array declarations
Valid1([0:x.EndNode])            As RealVariable;
Valid2(x.Interior)               As RealVariable;
Valid3([0]+x.Interior+x.EndNode) As RealVariable;
```

The following type of array declaration is not supported by the interpolation scheme:

```
//Declare domain
x As Domain;
//Declare set
MySet As IntegerSet([0:x.EndNode]);
```

```
//Invalid array declarations
Invalid1(MySet) As RealVariable;
Invalid2(MySet) As MyModel;
```

If the dependent distribution or array is within an array of sub models and the sub model array is dependent on the domain to be modified, the sub model array itself will represent one additional dimension.

For example:

```
Model MySub
  //Reference an external domain
  x As External Domain;
  //Declare distributions
  T1D As Distribution1D(X Is x) Of RealVariable;
  T2D As Distribution2D(X Is x, Y Is x) Of RealVariable;
  T3D As Distribution3D(X Is x, Y Is x, Z Is x) Of RealVariable;
  //Model equations
  :
  :
End


Model Main
  //Declare domain
  x As Domain;
  //Declare variables and distributions
  Tdist                 As Distribution1D(X Is x);
  Tarray([0:x.EndNode]) As RealVariable;
  //Declare array of submodels
  ModelArray([0:x.EndNode]) As MySub;
  //Model equations
  :
  :
End
```

In the above example the interpolation scheme will correctly interpolate distributions and arrays Tdist, Tarray, ModelArray.T1D and ModelArray.T2D. However ModelArray.T3D will not be interpolated as it contains 4 dependent dimensions.

The interpolation scheme will interpolate variable values, time derivatives, spatial derivatives and integrals

If the domain length is changed, the interpolation scheme will not extrapolate initial values for points beyond the length of the original mesh.

Interpolation of PDE Problems through Automation

To support automation of flowsheets through VB scripts and external applications such as Microsoft Excel™ the AMPDEInterp132.Automation object has been supplied. This object can be used to temporarily store information about the current problem and to execute the interpolation after manipulation of the domain of interest.

The properties exposed by this object are:

Name	Type	Read/Write	Description
Diagnostics	Boolean	Read/Write	Switch on diagnostic information that is sent to simulation messages window. The dependent distributions and arrays found will be listed.
Domain	ACM Object	Read/Write	Object reference to domain model of interest
DomainChanged	Boolean	Read	Has the domain changed significantly to require interpolation

The methods exposed by this are:

Name	Returns	Description
FindDependents	Integer	Must be called before any domain manipulation takes place. The method is used find all dependents on the previously defined domain and store variables values internally within the object. Returns the number of found dependents (active and inactive).
Interpolate	Boolean	Used to execute the interpolation of the previous domain mesh with respect to the new domain mesh configuration. Returns a flag to indicate whether interpolation was successful or not.

An example VB Script that manipulates the spacing preference of domain "x" inside the model "LaPlace2Ord2D" is as follows:

```
Option Explicit
Dim x
'Create an AMPDEInterp132.Automation object
Set x = CreateObject("AMPDEInterp132.Automation")
'Pass a reference to the domain of interest
Set x.Domain = LaPlace2Ord2D.x
'Turn on diagnostic output to simulation message window
```

```
x.Diagnostics = True
'Find all dependents on the domain
Application.Msg "Script: Found " & _
                CStr(x.FindDependents) & _
                " dependents on LaPlace2Ord2D.x"
'Reduce the current spacing preference by a third
With LaPlace2Ord2D.x
  .SpacingPreference.Value = 2/3*.SpacingPreference.Value
End With
'Has domain changed sufficiently to require interpolation?
If x.DomainChanged Then x.Interpolate
'Destroy object reference
Set x = Nothing
```

Using Distributions with PDE Modeling

Distributions are used to distribute a variable over a spatial domain. A distribution can reference one or more domains dependent on its dimensionality (1D, 2D, or 3D). In the modeling language, three built-in type definitions represent this concept:

- Distribution1D
- Distribution2D
- Distribution3D

The distribution sub-models that you declare in your process model represent distributed variables. They provide access to the values of the distributed variable at each point of the discretized spatial domain(s). To refer to the value of a particular distributed variable at a specific location (for example, node i in a 1D domain), use Distribution.value(i) or Distribution(i) for short. Using profile plotting, you can view these distributed variable values as your Y-axis variable along the length of the discretized domain. As your X-axis variable, use the Value array in your domain (for example, XDomain.Value([0:XDomain.EndNode]) or XDomain([0:XDomain.EndNode]) for short).

The built-in distribution models also contain the logic for calculating partial derivatives and integrals. More specifically, they calculate the derivatives and integrals of the distributed variables with respect to the independent spatial variables. First and/or second derivatives are calculated depending on the order of the PDE problem (as defined by the highest-order derivatives in the PDEs). Only first derivatives are considered for problems of order one. Both first and second derivatives are considered for problems of order two. The

exact form of the derivative calculation is based on the discretization method and order of approximation. For more information on partial derivative calculation, see Discretization Methods: Finite Difference and Discretization Methods: Finite Element Methods.

The built-in distribution models calculate integrals depending on the setting of the Integrals property. Single, double, and triple integrals can be computed for 1D, 2D, and 3D distributions, respectively. The exact form of the integral calculation is based on the discretization method(s) specified in the domain(s) associated with the distribution. For more information on integral calculation, see Switching On Integrals in Distribution Declarations.

Declaring Distributions for Distributed Variables

When modeling distributed parameter systems, you declare one or more distributed domains and associate each distributed variable in your model with up to three of these domains. The distributed variable is declared using a distribution. A distribution can refer only to domains that you have declared previously in your model.

Within a given process model, all variables do not necessarily have to be distributed over the same domain(s). For example, some variables may be distributed over both radial and axial domains, while others may be distributed over the axial domain only.

You can use an array to define a collection of distributions. For example, you can define a concentration array indexed by component names and distributed over a domain. For an example, see Using an Array of Distributions.

In the declaration of your distribution, you can use the IS operator to change the default domain sub-model names (that is, XDomain, YDomain, ZDomain) to the domain names of your choice.

You can also specify the highest order derivative of the distributed variable in each direction if it differs from the default set for the corresponding domain. To do this, assign a value to the HighestOrder*Derivative (where * is X, Y, or Z) property in the declaration of your distribution.

You can switch on integral calculations by using the Integrals property. With this property you control the calculation of single, double, and triple integrals.

In addition, you can define a distribution as hidden so that the model user does not see the properties for the distributed variable.

You can declare distributions in a process model using the following syntax.

Syntax for Declaring a Distribution

```
Distribution (Set , Set, ...), Distribution (Set…), ... AS
DistributionModel (DomainSubModel IS Domain, ... ,
                   Property : PropertyValue, ...) OF
VariableType;
```

Distribution..... Name you give to the distribution that you declare in your process model. You can instance a list of distributions using the same distribution model in one statement. Each distribution has all the properties defined in its distribution model. The value of each property will be the value assigned to the property in the distribution model type (that is, 1D, 2D, or 3D). You can modify the values of these properties.

Set ..... Optional set definition. Either a set name or a definition between square brackets [and]. Use a Set to define an array of distribution variables. To define multi-dimensional arrays, define a list of sets.

DistributionModel.. Name of a built-in distribution model. Valid built-in model names are DISTRIBUTION1D, DISTRIBUTION2D, and DISTRIBUTION3D.

DomainSubModel .. Name of a built-in domain sub-model declared in a built-in distribution model. The valid domain sub-model names are dependent on the distribution model. For DISTRIBUTION1D, the only valid DomainSubModel value is XDOMAIN. For DISTRIBUTION2D, the valid values are XDOMAIN and YDOMAIN. For DISTRIBUTION3D, the valid values are XDOMAIN, YDOMAIN, and ZDOMAIN.

Domain ..... Name you give to the domain in your process model. You can then refer to properties (variables and parameters) in the corresponding built-in domain and distribution model using the following syntax: Domain.PropertyName

Property..... Parameter or variable defined in the built-in distribution model. The following are distribution parameters that can be used in the assignment list:

You can assign a value to HighestOrder*Derivative (where * is X, Y, or Z) for a specific domain in the distribution. Valid values for these parameters are 0, 1 and 2. The default value is 1. No partial derivatives are calculated for a value of 0. Use a value of 0 to declare a variable that is distributed but that does not require partial derivatives. For a value of 1, only 1st-order partial derivatives are calculated. A value of 2 means that both 1st-order and 2nd-order partial derivatives are calculated.

HighestOrderXDerivative	Highest order partial derivative with respect to the independent spatial variable in the XDOMAIN. This parameter can be assigned for all three built in distribution models: DISTRIBUTION1D, DISTRIBUTION2D, and DISTRIBUTION3D.
HighestOrderYDerivative	Highest order partial derivative with respect to the independent spatial variable in the YDOMAIN. This parameter can be assigned for DISTRIBUTION2D and DISTRIBUTION3D.
HighestOrderZDerivative	Highest order partial derivative with respect to the independent spatial variable in the ZDOMAIN. This parameter can be assigned for DISTRIBUTION3D only.
Integrals	On/off switch for integral calculations. This property is a StringSet whose members are strings, for example ["idx", "idy"]. By default, the calculation of integrals is switched off and the Integrals property has a value of "None". This parameter can be assigned for all three built-in distribution models. For DISTRIBUTION1D, the valid value is "idx". For DISTRIBUTION2D, the valid values are "idx", "idy", "idxdy" or any combination of the three. For DISTRIBUTION3D, the valid values are

"idx", "idy", "idz", "idxdy", "idxdz", "idydz", "idxdydz" or any combination thereof.

VariableType..... Name of a defined variable type.

Examples of Declaring Distributions

All the following declarations appear within a process model definition.

In the first example, a temperature variable, T1, is distributed on a two-dimensional domain and takes on the characteristics of the built-in DISTRIBUTION2D model. In this example, the default value and the lower bound on the variable are altered:

```
T1 AS Distribution2D of Temperature (Value:373.0);
```

Two distributed temperature variables can be declared in one statement:

```
T1, T2 AS Distribution2D of Temperature (Value:373.0);
```

The next example shows how to define an array of distributed temperature variables. The variable T is indexed explicitly as having integer elements from 1 to 2:

```
T([1:2]) AS Distribution2D of Temperature (Value:373.0);
```

Use the IS operator to change the default domain names, XDomain and YDomain, to the more meaningful local names, Axial and Radial:

```
T([1:2]) AS Distribution2D(XDomain IS Axial, YDomain IS
Radial) of Temperature (Value:373.0);
```

Continuing with the preceding example, the Axial and Radial domains can provide second-order derivatives by assigning a value of 2 to HighestOrderXDerivative and HighestOrderYDerivative, respectively. These parameter values override the default value of 1 assigned in the built-in domain model. They also override any value assigned to HighestOrderDerivative in the declaration of the Axial and Radial domains:

```
T([1:2]) AS Distribution2D(XDomain IS Axial, YDomain IS
Radial, HighestOrderXDerivative:2,
HighestOrderYDerivative:2) of Temperature (Value:373.0);
```

To calculate both single and double integrals for a pressure variable P distributed in 2D, use the following StringSet syntax for the Integrals property:

```
P AS Distribution2D(Integrals:["idx", "idy", "idxdy"]) of
Pressure;
```

In the next example, the concentration variable, Conc, is declared as a distribution over three domains:

```
Conc AS Distribution3D of Conc_Mole;
```

The following example shows how to declare an array of concentration variables distributed over three domains. The variable Conc is indexed to the set called Components, which contains three string elements:

```
Components AS StringSet(["N2", "O2", "CO2"]);
```

```
Conc(Components) AS Distribution3D of Conc_Mole;
```

Declaring Distributed Variables that do not Require Partial Derivatives

The recommended way to declare a variable that is distributed but does not require partial derivatives is as follows:

```
Model Pipe

   XDomain as Domain (HighestOrderDerivative: 2);

   T as Distribution1D of Temperature;

   TWall as Distribution1D of Temperature
   (HighestOrderXDerivative: 0);

End
```

In the above example, the domain declaration sets the default order for all distributed variables to 2. This means that both ddx and d2dx2 are calculated for the distributed variable T. However, for TWall the default is overridden and set to zero. In this case the ddx and d2dx2 arrays are empty and no partial derivatives are calculated.

Alternatively you can simply index TWall over the same range as the domain:

```
Model Pipe

   XDomain as Domain(HighestOrderDerivative: 2);

   T as Distribution1D of Temperature;

   TWall([0:XDomain.EndNode]) as Temperature;

End
```

Switching On Integrals in Distribution Declarations

You can switch on integral calculation for a distributed variable using the Integrals property in the declaration of a distribution model. The Integrals property uses the following StringSet syntax:

Syntax for Switching On Integrals in Distribution Declarations

```
Distribution as DistributionModel (Integrals:[SetContents])
```

where *SetContents* is a valid comma-separated list of the contents of the Integrals property from the following table:

Distribution Model	Default	Single Integrals	Double Integrals	Triple Integrals
Distribution1D	"none"	"idx"	Invalid	Invalid
Distribution2D	"none"	"idx", "idy"	"idxdy"	Invalid

Distribution3D	"none"	"idx", "idy", "idz"	"idxdy", "idxdz", "idydz"	"idxdydz"

Integrals are switched off by default ("none").

You can use the following syntax to switch on single, double, and triple integrals for 1D, 2D, and 3D distributions, respectively:

```
T as Distribution1D (Integrals:"idx");

T as Distribution2D (Integrals:"idxdy");

T as Distribution3D (Integrals:"idxdydz");
```

The next example shows you how to switch on single integrals for a temperature variable T distributed in 1D. Note that CFD2 is used to calculate both partial derivatives and integrals for variable T distributed over XDomain.

```
XDomain as LengthDomain (DiscretizationMethod:"CFD2");

T AS Distribution1D(Integrals:"idx") of Temperature;
```

To compute single and double integrals for T distributed in 2D, use the following StringSet syntax for the Integrals property:

```
T AS Distribution2D(Integrals:["idx", "idy", "idxdy"]) of
Temperature;
```

The following example shows how to switch on triple integrals for a 3D distribution:

```
T AS Distribution3D(Integrals:"idxdydz") of Temperature;
```

Referring to Domains in Distributed Variable Declarations

Distributed variables can refer only to domains that you have declared previously in your model. For instance this sequence of declarations is legal:

```
Model Pipe

  // Declare distributed domain

  Axis as Domain of LengthParameter;

  // Declare array of concentration variables
  distributed over a 1D domain

  Conc(ComponentList) as Distribution1D(XDomain is
  Axis) of Conc_Mole (Value: 0.0);

End
```

whereas the following declarations are illegal because the Axis domain is used before it is declared:

```
Model Pipe
```

```
  // Declare array of concentration variables
  distributed over a 1D domain

  Conc(ComponentList) as Distribution1D(XDomain is
  Axis) of Conc_Mole (Value: 0.0);

  // Declare distributed domain

  Axis as Domain of LengthParameter;

End
```

Using an Array of Distributions

Use an array of distributions to model a property (for example, concentration) of various components distributed over a domain:

```
Model Pipe

  // Declare distributed domain

  Axis as Domain of LengthParameter;

  // Declare array of concentration variables
  // distributed over a 1D domain

  Conc(ComponentList) as Distribution1D(XDomain is
  Axis) of Conc_Mole (Value: 0.0);

  FeedConc(ComponentList) as Conc_Mole;

  V as Velocity;

  // PDE written over nodes 1 to EndNode

  $Conc(ComponentList) (Axis.Interior+Axis.EndNode)
  = -V * Conc(ComponentList)
  (Axis.Interior+Axis.EndNode).ddx;

  // Dirichlet boundary condition at left-hand
  // domain boundary

  Conc(ComponentList)(0) = FeedConc(ComponentList);

End
```

Notes:

- The declaration of the distributed array called Conc illustrates the convention for separating declaration qualifiers (for example, XDomain is Axis), which appear before the value type (for example, Concentration), and assignments (for example, Value:0.0), which are put after the value type.
- To refer to the concentration of a particular component (for example, "A") at a specific location (for example, node iX), use Conc("A").value(iX) or Conc("A")(iX) for short.

Referencing Domains and Distributions in Sub-Models

You can declare a domain and/or distribution in a model and then use this in a sub-model of that model. To do this you should declare the domain and/or distribution as External in the sub-model, and pass the name of the domain and/or distribution to be used when you instance the sub-model. For example to use the Domain X and distribution T in a sub-model called Conductivity, the main model would contain:

```
X as LengthDomain (Length:1, SpacingPreference:0.1);
T as Distribution1D (XDomain is X) of Temperature;
cond as conductivity (XDomain is X, YDomain is Y);
```

And the sub-model would define XDomain and T as external as follows:

```
Model Conductivity
XDomain as external Domain;
T as external Distribution1D;
```

XDomain and T can then be used in the Conductivity model as normal.

About IPDAE Models

The equation-based simulation environment enables you to perform dynamic simulations involving both lumped and distributed parameter processes. Distributed systems have properties that vary with one or more spatial dimensions as well as time. Both space and time are independent variables. These systems are described in terms of integral partial differential algebraic equations (IPDAE systems) expressed over a domain. In addition, these IPDAE systems usually involve both boundary conditions and initial conditions.

The general form of the IPDAE system over the bounded domain is:

$$f(t, x, \phi, \phi_t, \phi_x, \phi_{xx}) = 0$$

with boundary conditions of the following form:

$$g(t, x, \phi, \phi_t, \phi_x) = 0$$

Where:

t = Time, an independent variable.

x = Vector of at most three independent spatial variables.

ϕ = Vector of dependent variables of the form f(t,x).

ϕ_t = Time derivatives.

ϕ_x = First-order spatial derivatives.

ϕ_{xx} = Second-order spatial derivatives.

The partial derivatives with respect to the spatial variables are often written as follows:

$$\phi_x = \frac{\partial \phi(x,t)}{\partial x}, \quad \phi_{xx} = \frac{\partial^2 \phi(x,t)}{\partial x^2}$$

The well-known Method of Lines (MOL) is used to solve the time-dependent PDE systems. It is a two-stage process that discretizes in space and leaves the time domain continuous, thereby converting the IPDAE system to a DAE system with respect to time.

Partial Differential Equations (PDEs)

Partial differential equations contain one or more partial derivatives and involve at least two independent variables.

For distributed parameter modeling of physical and chemical processes, the most prevalent and useful PDEs are those with first- and/or second-order partial derivatives. Usually, at most four independent variables are involved, the three spatial coordinates and time.

A first-order, time-dependent PDE can be put in the form:

$$f = a\phi + b\phi_t + c\phi_x$$

Where:

t	=	Time, an independent variable.
x	=	Vector of at most three independent spatial variables.
ϕ	=	Vector of dependent variables of the form f(t,x).
ϕ_t	=	Time derivatives.
ϕ_x	=	First-order spatial derivatives.

In general, the coefficients can be functions of all the variables and their derivatives, for example:

$$a = a(t, x, \phi, \phi_t, \phi_x)$$

The following is an example of a widely-used, linear second-order PDE:

$$f = a\phi + b\phi_t + c\phi_x + d\phi_{xx}$$

Where *a, b, c*, and *d* are a function of (*t,x*).

To represent a distributed parameter problem fully, the model also requires auxiliary conditions. These are of two kinds, boundary conditions and initial conditions. The latter is used when time is one of the independent variables.

Writing PDEs in Models

Once you have declared one or more domains and distributions in your model, you can now start writing PDEs to describe complex physical phenomena such as mass, heat, and momentum transfer coupled to chemical reactions in 1D, 2D, and 3D.

The Aspen Custom Modeler modeling language for PDEs uses a number of common concepts throughout. The topics in this section describe the following concepts:

- Partial Derivatives
- Interior Set
- Open Domains
- Closed Domains
- Domain Slices

Specifying Partial Derivatives in PDEs

Specify partial derivatives in your PDE models using the dd* array for first derivatives and the d2d*2 array for second derivatives, where * can be x, y, or z depending on the dimensionality of your distribution.

Distribution Model	1st-Order Partial Derivatives	2nd-Order Partial Derivatives
Distribution1D	ddx	d2dx2
Distribution2D	ddx, ddy	d2dx2, d2dy2
Distribution3D	ddx, ddy, ddz	d2dx2, d2dy2, d2dz2

For one-dimensional distributions, access to first-order partial derivatives is via the ddx array defined in the built-in distribution models. First derivatives can appear in PDE model equations such as this advection equation:

```
$Temp(Axis.Interior+Axis.EndNode) = -V *
Temp(Axis.Interior+Axis.EndNode).ddx;
```

and in Neumann boundary conditions for 2nd-order PDE problems:

```
T(X.EndNode).ddx = 25.0;
```

The next example shows a LaPlace equation with second-order partial derivatives:

```
T(X.Interior,Y.Interior).d2dx2 +
T(X.Interior,Y.Interior).d2dy2 = 0.0;
```

Partial Derivative Type

The PartialDerivative type is a built-in variable type for use in custom PDE modeling. It has a reduced set of attributes (that is, value, description, and units) compared to a normal RealVariable. This saves memory if you are dealing with distributed parameter systems with a large number of partial derivatives.

Also refer to: *Specifying Partial Derivatives in PDEs*.

Using the Interior Set in PDEs

The Interior set enables you to write PDE models that are concise, correctly specified, and flexible from the standpoint of solution with different discretization methods. It is defined automatically in the built-in domain model and contains the grid points (nodes) required for your problem.

As shown in the table below, the contents of the Interior set are based on the following domain properties: order of your PDEs (HighestOrderDerivative), the discretization method you select (that is, finite difference vs. OCFE), and the number of sections and elements in your domain.

For 1st-order PDE problems the Interior set is always contiguous and contains all interior nodes, 1 to EndNode-1. For 2nd-order PDE problems however, the Interior set is contiguous only when using finite difference methods with a single section and OCFE methods with a single element. For all other 2nd-order PDE problems, the Interior set is not contiguous. In these cases, the model equation and 2nd-order derivative equation at the node(s) excluded from the set are replaced by a first-derivative continuity equation. For example, the Interior set does not include the section boundaries when using the finite difference methods for domains specified as having more than one section. For the OCFE methods, all element boundaries are excluded from the Interior set for 2nd-order PDE problems.

Discretization Methods	Highest Order Derivative	Number of Sections	Number of Elements per Section	Interior Set
BFD1, BFD2, CFD2, CFD4, FFD1, FFD2, UBFD4	1	---	---	[1:EndNode-1]
	2	=1	---	[1:EndNode-1]
	2	>1	---	[1:EndNode-1] – [Section Boundaries]
OCFE2, OCFE3, OCFE4	1	---	---	[1:EndNode-1]
	2	---	=1	[1:EndNode-1]
	2	---	>1	[1:EndNode-1] – [Element Boundaries]

- **Use the Interior set for writing (that is, indexing) your PDE model equations and initial conditions**. Failing to do so can lead to specification problems (that is, non-square systems) for 2nd-order PDE problems, especially when using finite difference methods with multiple sections or OCFE methods with multiple elements. Using the Interior set allows you to switch between finite difference and OCFE methods without having to modify your PDEs. The underlying requirement that finite difference and OCFE methods inherently require derivatives and model

equations to be evaluated at different discretization points is automatically handled.

- **Do not index your boundary conditions using the Interior set, but rather use an explicit set of contiguous nodes.** The reason is that any automatically excluded interior node in the Interior set (see the table above) leads to an underspecified problem. The following equations are examples of boundary conditions written using a contiguous set of nodes:
  ```
  T([1:X.EndNode-1], 0) = 0.0;
  T(0, [0:Y.EndNode]) = 0.0;
  ```
- **Do not index your integral equations using the Interior set, but rather use a set of contiguous nodes.** Since the Interior set is not always full (see the table above), indexing your integral equations using a set of contiguous nodes ensures that the integral for each individual element contributes to the calculation of the overall integral.

Using Interior Sets in Domain Sections

Each section in a domain contains its own Interior set. The union of the Interior sets for all sections in a domain is equal to the Interior set for the overall domain.

An example of how to access the value of a distributed variable using the Interior set for a domain section is shown here:

```
X As Domain (NumSections:2);
C As Distribution1D (XDomain Is x);
W As Distribution1D (XDomain Is x);

For i In X.Section(1).Interior Do
  W(i) = 2*C(i);
EndFor

For i In X.Section(2).Interior Do
  W(i) = 3*C(i)^4;
EndFor
```

Open Domains for PDE Modeling

The proper declaration of models of distributed parameter systems requires the accurate specification of the domain of applicability of each equation and boundary condition. You can easily distinguish between the interior of the domain and the different parts of the boundaries. The built-in domain model provides an integer set called Interior that can be used to facilitate the efficient writing of PDEs in your model.

An open domain is a domain without its boundaries. You can define an open domain in your model equations using the Interior set declared in the domain

model. As its name implies, the Interior set does not include the left-hand domain boundary [0] and right-hand [EndNode] domain boundary.

2nd-order PDEs are typically written over an open domain:

```
u(X.Interior,Y.Interior).d2dx2 +
u(X.Interior,Y.Interior).d2dy2 = 0.0;
```

Closed Domains for PDE Modeling

A closed domain is a domain with its boundaries included. The left-hand and right-hand domain boundaries are represented by nodes 0 and EndNode, respectively. You will often need to define a closed domain when writing your boundary conditions:

```
// Boundary condition for heat transfer to solid
$TS([0:X.EndNode]) = SCon * (TG([0:X.EndNode]) -
TS([0:X.EndNode]));
```

Important Note: Do not use the Interior set for writing boundary conditions, especially when using OCFE discretization methods and finite difference methods with multiple domain sections. You should always index boundary conditions explicitly using integer elements.

Using Domain Slices for PDE Modeling

The boundary condition equations in a model may involve just a sub-domain of an entire domain distribution. To define such sub-domains, you can make use of the built-in support for set operations. For example, in the case of the variable Temp which is distributed over both axial and radial domains, the notation Temp([0:Axial.EndNode], 0) refers to the temperature along the axis of the reactor:

```
Axial AS LengthDomain
 (DiscretizationMethod:"CFD4", Length: 10.0);
Radial AS LengthDomain
 (DiscretizationMethod:"CFD4", Length: 2.0);
Temp AS Distribution2D (XDomain is Axial,
 YDomain is Radial) of Temperature;
Temp([0:Axial.EndNode], 0) = 100.0;
```

Examples of Writing PDEs in Models

The first example shows the advection equation for flow in a tube where T is temperature and V is velocity. The PDE is 1st-order in time (t) and space (x). It is written over a domain that is open at the left-hand boundary. This is handled by using the Interior set and adding EndNode.

Advection Equation

$$\frac{\partial T}{\partial t} = -V \frac{\partial T}{\partial x}$$

```
// Modeling Language
$Temp = -V * Temp(Axis.Interior+Axis.EndNode).ddx;
```

For example, Fourier's second law for heat conduction in one-dimensional Cartesian coordinates:

Fourier's Second Law

$$\frac{\partial T}{\partial t} = \left(\frac{k}{\rho C_p} \right) \frac{\partial^2 T}{\partial x^2}$$

can be written in the modeling language as:

```
$T(X.Interior) = k/(rho*cp) * T(X.Interior).d2dx2;
```

Next, consider three well-known elliptic PDEs (zero-order in time) written over the interior of a domain defined in two-dimensional Cartesian coordinates:

LaPlace's Equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

```
// Modeling Language
u(X.Interior,Y.Interior).d2dx2 +
u(X.Interior,Y.Interior).d2dy2 = 0.0;
```

Poisson's Equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = \sinh(\sqrt{\pi^2 + 1}\, x) \sin(\pi y)$$

```
// Modeling Language
for ix in x.interior do
for iy in y.interior do
u(iX, iY).d2dx2 + u(iX, iY).d2dy2 =
sinh(sqrt(sqr(PI)+1.0)*X(iX)) * sin(PI*Y(iY));
endfor
endfor
```

Helmholtz's Equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = u$$

```
// Modeling Language

u(X.Interior,Y.Interior).d2dx2 +
u(X.Interior,Y.Interior).d2dy2 = u(X.Interior, Y.Interior);
```

Tip The range of each domain of every distributed variable in a PDE must be the same as shown in the following heat transfer equation:

```
// Heat transfer equation: Time-dependent second order PDE.

// PDE is specified over the domain's interior.

$T(X.Interior, Y.Interior) = D *     T(X.Interior,
Y.Interior).d2dx2 + D * T(X.Interior, Y.Interior).d2dy2;
```

On the other hand, the following PDE example is not legal because the range of the X domain for the $T variable is not consistent with the range of X domain for the distributed variables on the right-hand side of the equation:

```
// Heat transfer equation: Time-dependent second order
partial differential

// equation. PDE is specified over the interior of the
domain.

$T([0]+X.Interior, Y.Interior) = D * T(X.Interior,
Y.Interior).d2dx2 + D * T(X.Interior, Y.Interior).d2dy2;
```

Using PDEs with a Conservative Term

As an example, consider the following heat transfer equation within a cylinder of infinite length:

$$\frac{\partial T}{\partial t} = \frac{a}{r}\frac{\partial}{\partial r}\left(r\frac{\partial T}{\partial r}\right) + \frac{q}{c_p}$$

where *T* is temperature, *r* is radial position, *a* is the thermal diffusivity, *q* is the heat duty, and C_p is the heat capacity. Due to the use of the conservative term:

$$\frac{\partial}{\partial r}\left(\frac{r\partial T}{\partial r}\right)$$

it is necessary to introduce a distribution for temperature and another one for the inner radial flux expression (for example, rdTdr). For the radial distribution of T over r, we need first to create a Domain. So we declare:

```
r as LengthDomain(DiscretizationMethod:"CFD4");
```

For the temperature, we need a 1D distribution:

```
T as Distribution1D(XDomain is r) of temperature;
```

To handle the conservative term we need to create an additional distribution:

```
rdTdr as Distribution1D (XDomain is r) of realvariable;
```

The heat conduction equation with cylindrical coordinates can be written as follows:

```
for i in r.Interior do
  $T(i) = a/r(i)*rdTdr(i).ddx + q/cp;
endfor
```

Finally, we need the equation to define the additional distribution for the conservative term:

```
rdTdr = r*T.ddx;
```

Using Boundary and Initial Conditions for PDE Modeling

The solution of PDEs must satisfy certain auxiliary conditions (initial and boundary) on the boundary of the specified domain. These auxiliary conditions of a PDE system determine its unique solution from among an infinite number of solutions. They serve as the starting point for calculating the interior solution. The boundary conditions are a fundamental part of the description of the process system behavior, while the initial conditions define the initial state of the process system and may often differ from one simulation to the next.

In general, derivatives with respect to the spatial independent variables must be at least one order lower than the highest-order derivative in the PDE. Thus, for a 2nd-order PDE in x, the auxiliary conditions can be no higher than 1st-order:

$$g(t, x, \phi, \phi_t, \phi_x) = 0$$

where:

t	=	Time, an independent variable.
x	=	Vector of at most three independent spatial variables.
ϕ	=	Vector of dependent variables of the form f(t,x).
ϕ_t	=	Time derivatives.
ϕ_x	=	First-order spatial derivatives.

The auxiliary conditions associated with 2nd-order PDEs typically involve values of the dependent variable and/or its first derivatives at particular locations and time. In these cases, the preceding general form becomes:

$$f = a\phi + b\phi_x$$

where *a, b*, and *f* are a function of (*t,x*). The derivative is normal to the domain boundary.

In the mathematical literature, three of the most commonly recognized conditions are:

- Dirichlet
- Neumann
- Cauchy

The boundary conditions arising in your applications may be significantly more complicated than the Dirichlet, Neumann, and Cauchy classifications discussed here.

Important:

- **When writing initial conditions, use the Interior set** so that your initial specifications are only applied to the Interior points. This avoids over-specification for second-order PDE problems with time derivatives. This is particularly important when using OCFE discretization methods or finite difference methods with multiple domain sections.
- **When writing boundary conditions, do not use the Interior set** as you typically want them to apply to contiguous locations along the boundary. Remember here that for second-order PDE problems solved using OCFE discretization methods or finite difference methods with multiple domain sections, the Interior set does not contain the contiguous set of nodes from 1 to EndNode-1 (See "Using the Interior Set in PDEs" for more detailed information). In view of this, you should **always index boundary conditions explicitly using a set of contiguous nodes**.
- **When writing boundary conditions involving second partial derivatives, you may need to explicitly write the derivative equations and include them in your model.** The built-in domain models do not include all second derivatives at the domain boundaries. For more information, see Boundary Conditions Containing Second-Order Derivatives.

Dirichlet conditions (b=0 in the preceding equation) specify the values of the dependent variables at the boundary points as a function of time. These Dirichlet points represent the exact solutions at the boundary points.

Neumann conditions (a=0 in the preceding equation) specify the values of the first-order spatial derivatives on the boundary. This corresponds to specifying the flux across the boundary. If f=0, we have a homogeneous Neumann boundary condition, which leads to property conservation within the domain.

Cauchy

Cauchy or mixed boundary conditions comprise property flux and property value at the boundary ($a, b \neq 0$ in the preceding equation). This is often used to specify property transport over the boundary (for example, interface kinetic, oxidizing surface conditions).

Example of Initial and Boundary Conditions

The following model gives examples of Dirichlet and Neumann boundary conditions, as well as initial conditions:

```
Model HeatEquation

// Heat conduction example
```

```
// Domain declaration
 X as LengthDomain(DiscretizationMethod:"OCFE4",
                   HighestOrderDerivative: 2);
// Declaration of distributed temperature variable
 T as Distribution1D(XDomain is X) of Temperature;
// Declaration of thermal diffusivity for the solid
 K as RealParameter(1/32);


// PDE - open on both ends of the domain as boundary
// conditions specified at 0 and EndNode
 $T(X.Interior) = k*T(X.Interior).d2dx2;


// Initial conditions
 T([1: X.EndNode-1]): 10.0;
// Initial specification must only be applied to 'Interior'
// points to avoid over-specification when using OCFE
methods,
// or finite difference methods with multiple sections
 T(X.Interior): initial;


// Dirichlet boundary condition at left-hand domain
boundary
 T(0) = 0.0;


// Neumann boundary condition at right- hand domain
boundary
 T(X.EndNode).ddx = 25.0;


End
```

Boundary Conditions Containing Second-Order Derivatives

If your model requires 2nd-order derivatives at a domain boundary and they are not provided automatically by the Distribution model, you must explicitly write the 2nd-order derivative equations and include them in your model. The built-in distribution models explicitly exclude some 2nd-order partial derivatives at the domain boundaries for the following reasons:

- The majority of process models do not involve boundary conditions containing 2nd-order derivatives.
- The OCFE discretization methods do not require 2nd-order derivative calculations at element boundaries including the domain boundaries. Excluding the unnecessary 2nd-order derivatives at the boundaries saves on wasted operations and memory and also allows Aspen Custom Modeler to provide more useful analysis for specification errors.

The table below presents a list of the 2nd-order derivatives that are calculated by the distribution models.

Distribution Model	2nd-Order Partial Derivatives
Distribution1D	d2dx2 (XDomain.Interior)
Distribution2D	d2dx2 (XDomain.Interior, [0 + YDomain.Interior + YDomain.EndNode]) d2dy2([0 + XDomain.Interior + XDomain.EndNode], YDomain.Interior)
Distribution3D	d2dx2 (XDomain.Interior, [0 + YDomain.Interior + YDomain.EndNode], [0 + ZDomain.Interior + ZDomain.EndNode]) d2dy2([0 + XDomain.Interior + XDomain.EndNode], YDomain.Interior, [0 + ZDomain.Interior + ZDomain.EndNode]) d2dz2([0 + XDomain.Interior + XDomain.EndNode], [0 + YDomain.Interior + YDomain.EndNode], ZDomain.Interior)

Second-Order Derivatives for Use in Full Discretization

The table above shows that while Distribution2D does not calculate d2dx2 for nodes ([0 + XDomain.EndNode], *), it does calculate them for boundary nodes (*, [0 + YDomain.EndNode]). The reason for providing the 2nd-order partial derivatives for d2dx2 at the YDomain boundaries and d2dy2 at the XDomain boundaries for Distribution2D is to allow you to set up a full discretization case where one domain is a spatial domain and the other is a time domain. In this case, you can write PDEs containing 2nd-order partial derivatives over the entire time domain, including at the initial and final time. The same capability is provided for Distribution3D where any of the three coordinates, x, y, or z can be used as a time domain.

Writing Second-Order Derivatives at Domain Boundaries

As discussed above, if your model requires second derivatives at the domain boundaries, you must explicitly write equations for the second derivatives and include them in your model. For example, you can write standard second-order forward and backward finite difference equations for the left-hand boundary and right-hand boundary, respectively.

This example shows a second-order forward finite difference equation that can be used as a boundary condition for the left-hand domain boundary. The second-partial derivative of a temperature variable, T, is set to zero.

```
(T(0)-2*T(1)+T(2)) / (X.value(1)-X.value(0))^2 = 0.0;
```

The next example shows a 2nd-order backward finite difference equation that can be used as a boundary condition for the right-hand domain boundary. The second-partial derivative of a temperature variable, T, is set to zero.

```
(T(X.EndNode-2)-2*T (X.EndNode-1)+T (X.EndNode)) /
(X.value(X.EndNode)-X.value(X.EndNode-1))^2 = 0.0;
```

If your boundary conditions contain second derivatives which are not automatically computed in the distribution model and not included in the derivative equations in your model, Aspen Custom Modeler® gives an error indicating that your model has equations that use variables that do not exist. The error message is displayed in the Simulation Messages window when checking the syntax of your model.

For example, consider the following HeatEquation model with a boundary condition involving a second-order partial derivative:

```
Model HeatEquation


// Heat conduction example
// Domain declaration
 X as LengthDomain(DiscretizationMethod:"OCFE4",
                   HighestOrderDerivative: 2);
// Declaration of distributed temperature variable
  using a Distribution1D model
 T as Distribution1D(XDomain is X) of Temperature;
// Declaration of thermal diffusivity for the solid
 K as RealParameter(1/32);


// PDE - open on both ends of the domain since boundary
// conditions specified at 0 and EndNode
 $T(X.Interior) = k*T(X.Interior).d2dx2;


// Initial conditions
 T([1: X.EndNode-1]): 10.0;
// Initial specification must only be applied to 'Interior'
// points to avoid over-specification when using OCFE
methods,
// or finite difference methods with multiple sections
 T(X.Interior): initial;


// Dirichlet boundary condition at left- hand domain
boundary
 T(0) = 0.0;


// Boundary condition containing second partial derivative
```

```
// Applied at right-hand domain boundary
 T(X.EndNode).d2dx2 = 0.0;


End
```

The following syntax error message is displayed in the Simulation Messages window when Aspen Custom Modeler compiles the PDE model shown above:

```
Model HeatEquation has equations that use variables which
do not exist:

HeatEquation.T.d2dx2(8) - Failed path is (8)
```

To avoid this error, you can replace the boundary condition involving d2dx2 with a polynomial approximation as discussed above.

Writing Integrals in PDE Models

Integrals occur frequently in PDE models for process engineering, especially in application areas that require population balance modeling (e.g., crystallization).

The calculation of integrals for a distributed variable is switched on by using the Integrals property available in the distribution models.

The discretization method specified in the domain model for calculating partial derivatives is also used to calculate the integral over each element in the distribution.

Specifying Integrals

You can specify integrals in your PDEs using the id* array for single integrals, the id*d* array for double integrals, and the idxdydz for triple integrals, where * can be x, y, or z depending on the dimensionality of your distribution. The following table shows valid values for integrals.

Distribution Model	Single Integrals	Double Integrals	Triple Integrals
Distribution1D	idx	Not Applicable	Not Applicable
Distribution2D	idx, idy	idxdy	Not Applicable
Distribution3D	idx, idy, idz	idxdy, idxdz, idydz	idxdydz

The indices of the integral arrays range from 0 to EndNode. The convention is that the integral over element *i-1* to *i* is stored in location *i* in the integral array. By default, the integral value for node 0 is 0.0. For example, the integral for the 1D variable T over the element *i-1* to *i* is stored in T(*i*).idx. By default, T(0)=0.0.

Since an integral is calculated over each individual element in a distribution, you can compute the integral over the entire distribution by summing up the individual contributions using the SIGMA operator as shown here:

```
Integral = SIGMA(T.idx);
```

The above equation is shorthand for:

```
Integral = SIGMA(T([0:X.EndNode]).idx);
```

Note that the summation above is done from 0:X.EndNode and not over the Interior set since we need to sum up the integrals over all elements. If the Interior set is used, the calculated integral is incorrect when using an OCFE method since the Interior set is not full (i.e., does not contain all nodes).

Examples of Writing Integral Equations

The The following example shows the use of an integral to calculate of the average temperature over a 1D distribution:

```
X as LengthDomain(Length:10);

T as Distribution1D(XDomain is X, Integrals: ”idx”) of
Temperature;

Tavg as Temperature;

Tavg = SIGMA(T.idx)/X.Length;
```

The previous example can be extended to 2D in the following way:

```
X, Y as LengthDomain(Length:10);

T as Distribution2D(XDomain is X, YDomain is Y, Integrals:
”idxdy”) of Temperature;

Tavg as Temperature;

Tavg = SIGMA(T.idxdy)/(X.Length*Y.Length);
```

This example uses an integral to computes the average temperature over a slice (i.e., left-hand edge) of a 2D distribution:

```
X, Y as LengthDomain;

T as Distribution2D (XDomain is X, Ydomain is Y,
Integrals:”idy”) of Temperature;

Tavg as Temperature;

Tavg = SIGMA(T(0,[0:Y.EndNode]).idy) / Y.Length;
```

In another 2D example, in this case a jacketed tubular reactor model, the energy balance on the cooling medium looks like:

```
Axial as LengthDomain;

Radial as LengthDomain;

T as Distribution2D(XDomain is Axial, YDomain is Radial,
Integrals:"idx") of Temperature;

Q = U*S*(SIGMA(T([0:Axial.EndNode], Radial.EndNode).idx) /
Axial.Length - Tcw);
```

The following crystallization example shows how the magma density is calculated using an integral over the crystal size domain. The integral involves two distributed variables, one for the crystal volumetric shape factor and one for the exponential of log crystal population density:

```
//Domain for crystal size
CSize as LengthDomain;
// Log crystal number density
LogN as Distribution1D(XDomain is CSize) of notype;
// Exponential of log crystal number density
eLogN as Distribution1D(XDomain is CSize, Integrals: "idx")
of notype;
// Kv is crystal volumetric shape factor
Kv as Distribution1D(XDomain is CSize, Integrals: "idx") of
pos_small;
// Magma density
Mt as conc_mass;
// Solid density
RhoS as RealParameter(2660.0);
eLogN = EXP(LogN);
// Definition of magma density
Mt = RhoS * 1E-12 * SIGMA(Kv.idx * CSize^3 *
eLogN.idx)/.015;
```

The following example demonstrates the use of integrals for a system of cylindrical geometry (z, r), whereby the average temperature is evaluated by:

$$T_{avg} = \frac{2\pi \int_0^L \int_0^R T(z,r).r.dr.dz}{\pi R^2 L}$$

A sample model is as follows:

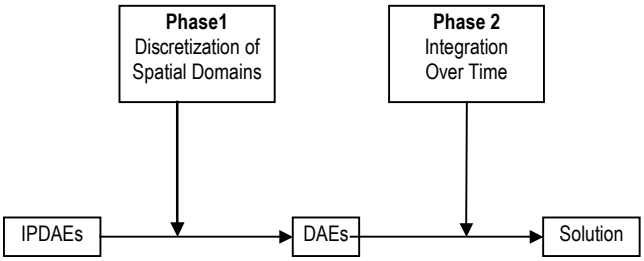
```
Model CylindricalTemp
  Radius as lengthparameter(1);
  L      as lengthparameter(2);
  pi     as realparameter(4*atan(1));
  r      as lengthdomain(length:Radius);
  z      as lengthdomain(length:L);
```

```
  T      as distribution2D(XDomain is r, Ydomain is z);
  Tr     as distribution2D(XDomain is r, Ydomain is z,
integrals:"idxdy");
  Tavg   as realvariable;
  Tavg2  as realvariable;
  for ir in [0:r.EndNode] do
    for iz in [0:z.EndNode] do
      Tr(ir, iz) = T(ir, iz)*r(ir);
    endfor
  endfor
  for ir in [0:r.EndNode] do
    for iz in [0:z.EndNode] do
      T(ir, iz) = 3*r(ir);
    endfor
  endfor
  // Tavg = 2*pi*integral integral [T(r,z)*r*dr*dz] /
volume
  Tavg = 2*pi*sigma(Tr.idxdy)/ (pi*Radius^2*L);
End
```

Using Method of Lines for the Solution of PDEs

The Method of Lines (MOL) solves time-dependent PDE systems by discretizing in space, but leaving the time variable continuous (rather than time evaluated at discrete nodes). If you think of the solution as a surface over the space-time plane, the MOL computes cross-sections of the surface along series of lines, each parallel to the time axis and corresponding to one of the discrete spatial nodes.

The MOL solution process has two phases. In the first, the partial derivatives and integrals with respect to the independent spatial variables are replaced with algebraic approximations (finite difference or finite element) evaluated at discrete points. In the second phase, a series of DAE systems are solved using a dynamic integrator (See the Integrator tab of the Solver Properties dialog box for a list of integrators available). The accuracy of the approximate solution depends on the element spacing and the integration step size in time.



Two Key Steps in MOL Approach

1 Replace continuous domain(s) of equations by a discrete grid of points (nodes) and replace derivatives and integrals in PDEs by finite difference or finite element approximations.

2 Obtain numerical solution at each time step that is a table of approximate values at selected nodes in spatial domain(s).

Finite Difference Methods: General

In the finite difference method, the continuous problem domain is discretized in terms of the values of the dependent variables at discrete points only. First-order and second-order partial derivatives are approximated by finite difference formulas derived using Taylor series expansions. The general forms of these derivative equations are:

$$\frac{\partial \phi}{\partial x} = \left(\frac{1}{ah}\right) A^{[n]]} \phi$$

$$\frac{\partial^2 \phi}{\partial x^2} = \left(\frac{1}{bh}\right) B^{[n]]} \phi$$

Where:

$\frac{\partial \phi}{\partial x}$ = Derivative vector.

ϕ = Dependent variable vector.

x = Independent spatial variable.

h = element spacing.

1/*a*,1/*b* = multiplying constant.

A, B = (n+1)x(n+1) differentiation matrix. The elements of the matrix are the weighting coefficients of nth-order approximations at the n+1Taylor series points.

n = order of approximation.

Finite difference methods differ in terms of type (for example, forward, backward, central, upwind-biased) and order of approximation.

Discretization Methods: Finite Difference

You can choose between several finite difference options for use as a domain discretization method. The default method is BFD1. When writing your PDE model, you can specify a different DiscretizationMethod property when declaring a domain. You can also select another method using the Domain Configuration Form.

Method	Order of Approximation	Type of FD Approximation
BFD1	1st-order	Backward
BFD2	2nd-order	Backward
CFD2	2nd-order	Central
CFD4	4th-order	Central
FFD1	1st-order	Forward
FFD2	2nd-order	Forward
UBFD4	4th-order	Upwind-Biased

Remarks

The finite difference methods are based on the assumption of uniform grids.

1st-Order Backward Finite Difference (BFD1)

PDE Order	LHB Node 0	Section1 Node 1	Section 1 Node 2	Section 1 Node 3	Sbound Node 4	Section 2 Node 5	Section 2 Node 6	Section2 Node 7	RHB EndNode
1-ddx	FFD1	BFD1	BFD1	BFD1	BFD1	BFD1	BFD1	BFD1	BFD1
2-d2dx2	None	CFD2	CFD2	CFD2	ddx Cont	CFD2	CFD2	CFD2	None

2nd-Order Backward Finite Difference (BFD2)

PDE Order	LHB Node 0	Section 1 Node 1	Section 1 Node 2	Section 1 Node 3	Sbound Node 4	Section 2 Node 5	Section 2 Node 6	Section 2 Node 7	RHB EndNode
1-ddx	FFD2	CFD2	BFD2	BFD2	BFD2	CFD2	BFD2	BFD2	BFD2
2-d2dx2	None	CFD2	BFD2	BFD2	ddx Cont	CFD2	BFD2	BFD2	None

2nd-Order Central Finite Difference (CFD2)

PDE Order	LHB Node 0	Section 1 Node 1	Section 1 Node 2	Section 1 Node 3	Sbound Node 4	Section 2 Node 5	Section 2 Node 6	Section 2 Node 7	RHB EndNode
1-ddx	FFD2	CFD2	CFD2	CFD2	BFD2	CFD2	CFD2	CFD2	BFD2
2-d2dx2	None	CFD2	CFD2	CFD2	ddx Cont	CFD2	CFD2	CFD2	None

4th-Order Central Finite Difference (CFD4)

PDE Order	LHB Node 0	Section 1 Node 1	Section 1 Node 2	Section 1 Node 3	Section 1 Node 4	Sbound Node 5	Section 2 Node 6	Section 2 Node 7	Section 2 Node 8	Section 2 Node 9	RHB EndNode
1-ddx	FFD4	DBFD4	CFD4	CFD4	UBFD4	BFD4	DBFD4	CFD4	CFD4	UBFD4	BFD4
2-d2dx2	None	DBFD4	CFD4	CFD4	UBFD4	ddx Cont	DBFD4	CFD4	CFD4	UBFD4	None

1st-Order Forward Finite Difference (FFD1)

PDE Order	LHB Node 0	Section 1 Node 1	Section 1 Node 2	Section 1 Node 3	Sbound Node 4	Section 2 Node 5	Section 2 Node 6	Section 2 Node 7	RHB EndNode
1-ddx	FFD1	FFD1	FFD1	FFD1	BFD1	FFD1	FFD1	FFD1	BFD1
2-d2dx2	None	CFD2	CFD2	CFD2	ddx Cont	CFD2	CFD2	CFD2	None

2nd-Order Forward Finite Difference (FFD2)

PDE Order	LHB Node 0	Section 1 Node 1	Section 1 Node 2	Section 1 Node 3	Sbound Node 4	Section 2 Node 5	Section 2 Node 6	Section 2 Node 7	RHB EndNode
1-ddx	FFD2	FFD2	FFD2	CFD2	BFD2	FFD2	FFD2	CFD2	BFD2
2-d2dx2	None	FFD2	FFD2	CFD2	ddx Cont	FFD2	FFD2	CFD2	None

4th-Order Upwind Biased Finite Difference (UBFD4)

PDE Order	LHB Node 0	Section 1 Node 1	Section 1 Node 2	Section 1 Node 3	Section 1 Node 4	Sbound Node 5	Section 2 Node 6	Section 2 Node 7	Section 2 Node 8	Section 2 Node 9	RHB EndNode
1-ddx	FFD4	DBFD4	CFD4	UBFD4	UBFD4	BFD4	DBFD4	CFD4	UBFD4	UBFD4	BFD4
2-d2dx2	None	DBFD4	CFD4	UBFD4	UBFD4	ddx Cont	DBFD4	CFD4	UBFD4	UBFD4	None

Discretization Methods: Finite Element Methods

The finite element family of methods divides the domain into many smaller elements and applies weighted residual methods within each element. The weighted residual methods assume that the solution can be represented as a weighted combination of known polynomial approximations of order n with unknown weighting coefficients. The latter are determined to satisfy boundary conditions and the system of equations at a finite set of points, called collocation or polynomial points. The precise criterion used to choose the weighting coefficients determines the method.

The weighted residual method that is combined with the finite element concept is the method of orthogonal collocation (OCFE). Second-, third-, and fourth-order OCFE methods are provided as shown in the following table. You can specify that you want to use an OCFE method using the DiscretizationMethod property assignment when declaring a domain.

Method	Order of Approximation	Type of Approximation	Polynomial (Collocation) Points
OCFE2	2nd-order	Orthogonal Collocation on Finite Elements	0.00000, 0.50000, 1.00000
OCFE3	3rd-order	Orthogonal Collocation on Finite Elements	0.00000, 0.21132, 0.50000, 0.78868, 1.00000
OCFE4	4th-order	Orthogonal Collocation on Finite Elements	0.00000, 0.11270, 0.50000, 0.88730, 1.00000

As shown in the following tables for OCFE2, OCFE3, and OCFE4, the domain is divided into a number of elements and orthogonal collocation is applied within each element. The value of EndNode is equal to the number of elements times the order of the method, where the order is the number of collocation points plus one.

By using the Interior set provided, the PDEs in your model are evaluated automatically at the interior collocation points only. The correct derivatives are also evaluated automatically.

For 1st-order PDE problems, the first derivatives at the element boundaries are calculated using the element to the left of the boundary. For 2nd-order PDEs, the OCFE methods are implemented using the condition that the first derivatives are continuous between elements. This treatment of the boundaries between elements gives a solution that is continuous, with continuous flux, as in the exact solution.

OCFE2

PDE Order	LHB Node 0	Element 1 Coll 1	ElemBound Node 2	Element2 Coll 1	RHB EndNode 4
1	ddx	ddx	ddx Elem1	ddx	ddx
2	None	d2dx2	ddx Cont	d2dx2	None

OCFE3

PDE Order	LHB Node 0	Element 1 Coll 1	Element 1 Coll 2	ElemBound Node 3	Element 2 Coll 1	Element 2 Coll 2	RHB EndNode 6
1	ddx	ddx	ddx	ddx Elem1	ddx	ddx	ddx
2	None	d2dx2	d2dx2	ddx Cont	d2dx2	d2dx2	None

OCFE4

PDE Order	LHB Node 0	Element1 Coll 1	Element1 Coll 2	Element1 Coll 3	ElemBound Node 4	Element2 Coll 1	Element2 Coll 2	Element2 Coll 3	RHB EndNode 8
1	ddx	ddx	ddx	ddx	ddx Elem1	ddx	ddx	ddx	ddx
2	None	d2dx2	d2dx2	d2dx2	ddx Cont	d2dx2	d2dx2	d2dx2	None

Finite Element Remarks

One advantage of finite element methods is their ability to handle problems with steep gradients by concentrating many small elements in areas with steep gradients.

Mixer-Reactor-Absorber (PDE) Example Description

This example demonstrates the use of PDE modeling in the dynamic simulation of an isothermal process flowsheet consisting of a well-stirred mixing tank, a tubular reactor, and a gas absorber.

Components A and B at a concentration ratio of 2 to 1 in the fresh feed stream enter the mixer along with the contents of recycle stream. The mixer outlet stream is sent to the reactor which carries out the gas-phase reaction: $A + B \rightarrow 2C$. The reactor product enters the bottom of the countercurrent absorber where C is partially absorbed in the liquid phase. The remaining gas is recycled to the mixing tank.

The dynamic simulation requires the simultaneous solution of the ordinary differential equations (ODEs) describing the mixer and the PDAEs for the tubular reactor and gas absorber.

This example is taken from the paper by M. Oh and C.C. Pantelides, A modeling and simulation language for combined lumped and distributed parameter systems, Computers & Chemical Engineering, Volume 20, Issues 6/7, pp. 611-633 (1996).

Also, refer to the section: *Running the Mixer-Reactor-Absorber (PDE) Example*

Running the Mixer-Reactor-Absorber (PDE) Example

Example simulations are included in your Aspen Custom Modeler installation. If you have installed in the default location, the files for this example will be in the folder:

```
C:\Program Files\AspenTech\Aspen Custom Modeler
12.1\Examples\MixerReactorAbsorber
```

To prepare to run this example:

- Copy the files in the example directory to a convenient working folder, for example:

```
C:\Program Files\AspenTech\Working Folders\Aspen Custom
Modeler 12.1\MixerReactorAbsorber
```

To run this example, follow these steps:

1. Double-click the file **MixerReactorAbsorber.acmf** to open it.
2. In the All Items pane of the Simulation Explorer, ensure Flowsheet is selected.
3. In the Contents pane, double-click on Blocks icon.
4. In the Contents pane, double-click the Absorber icon.
5. In the Contents pane, double-click the plot icon to display it.
6. Repeat steps 4 and 5 for the Mixer and Reactor.
7. Ensure that the run mode is Dynamic, and then run the simulation.

The 3D profile plots provided show the concentration profiles of component A in the reactor and gas absorber as a function of the spatial and temporal domains. The 2D plot for the mixer shows the time transient concentration of component A.

Jacketed Reactor (PDE and Integral) Example Description

In this example, a fixed-bed, jacketed tubular reactor is used to carry out the oxidation of o-xylene (component A) to phthalic anhydride (component B). The top-level cooled reactor model contains a pseudo-homogeneous 2D PDE sub-model for the tubular reactor and a cooling jacket sub-model that assumes perfect mixing and constant holdup. The two sub-models are linked by equating the reactor wall temperature to that of the cooling medium, and expressing the total heat input to the jacket as an integral of the heat loss from the tube surface over the entire tube length. The resistance to heat transfer occurs primarily between the reactor contents and tube wall. The wall of the tube is essentially at the cooling medium temperature.

This example is taken from the paper by M. Oh and C.C. Pantelides, A modeling and simulation language for combined lumped and distributed parameter systems, Computers & Chemical Engineering, Volume 20, Issues 6/7, pp. 611-633 (1996).

Also, refer to the section: *Running the Jacketed Reactor (PDE and Integral) Example.*

Running the Jacketed Reactor (PDE and Integral) Example

Example simulations are included in your Aspen Custom Modeler installation. If you have installed in the default location, the files for this example will be in the folder:

```
C:\Program Files\AspenTech\Aspen Custom Modeler
12.1\Examples\JacketedReactor
```

To prepare to run this example:

- Copy the files in the example directory to a convenient working folder, for example:

```
C:\Program Files\AspenTech\Working Folders\Aspen Custom
Modeler 12.1\JacketedReactor
```

To run this example, follow these steps:

1. Double-click the file **JacketedReactor.acmf** to open it.
2. In the All Items pane of the Simulation Explorer, ensure Flowsheet is selected.
3. In the Contents pane, double-click on Blocks icon.
4. In the Contents pane, double-click the CooledReactor icon.
5. In the Contents pane, double-click on the three plot icons to display them.
6. Ensure that the run mode is Dynamic, and then run the simulation.

The two 3D profile plots show the concentration profiles of component B and the temperature profiles at the center of the tubular reactor over time. The 2D plot presents the cooling load and temperature of the cooling medium over time.

Crystallizer (PDE and Integral) Example Description

This example considers a continuous mixed suspension mixed product removal (CMSMPR) crystallization unit that operates in cooling mode to produce crystals from aqueous solutions. The crystallizer temperature is assumed to be constant and any crystal breakage is ignored. The IPDAE model consists of the mass balance (ODE), crystal population balance (PDE), integral equation for magma density, and several algebraic equations.

The dynamic simulation considers the start-up of the crystallizer, with the unit initially full of pure water (i.e., the initial liquid concentration of solute and the logarithmic number density for all non-zero crystal sizes are set to zero. The liquid phase concentration remains below the saturation line for the initial

0.77 hours of operation and then rises towards the feed value (0.15 kg/kg) until the rate of crystallization increases sufficiently to reverse this trend.

This example is taken from the Ph.D. Thesis by M. Oh, *Modelling and Simulation of Combined Lumped and Distributed Processes, Imperial College of Science, Technology and Medicine, London* (1995).

Also refer to; *Running the Crystallizer (PDE and Integral) Example*.

Running the Crystallizer (PDE and Integral) Example

Example simulations are included in your Aspen Custom Modeler installation. If you have installed in the default location, the files for this example will be in the folder:

```
C:\Program Files\AspenTech\Aspen Custom Modeler
12.1\Examples\Crystallizer
```

To prepare to run this example:

- Copy the files in the example directory to a convenient working folder, for example:

```
C:\Program Files\AspenTech\Working Folders\Aspen Custom
Modeler 12.1\Crystallizer
```

To run this example, follow these steps:

1. Double-click the file Crystallizer.acmf to open it.
2. In the All Items pane of the Simulation Explorer, ensure Flowsheet is selected.
3. In the Contents pane, double-click on Blocks icon.
4. In the Contents pane, double-click the Crystallizer icon.
5. In the Contents pane, double-click the three plot icons to display them.
6. Right-click on the profile plot for Crystal_Population_Density and select Properties.
7. Click on the Contours tab and change the Number of levels to 10.
8. Click OK.
9. Ensure that the run mode is Dynamic, and then run the simulation.

The 3D profile plot shows the crystal population density profiles in the crystallizer as a function of the crystal size and time. The 2D plots show the time transients for the magma density and the liquid phase solute concentration.

PDE Modeling Glossary

Dirichlet

Dirichlet is a type of boundary condition that specifies the values of the dependent variables at the boundary points as a function of time.

Distributed parameter system

A distributed parameter system contains process variables that vary with respect to spatial position as well as time.

Distribution

A distribution is a variable distributed over one or more domains.

Domain

A domain is a region over which a distributed parameter system is discretized.

Method of Lines

The Method of Lines is an approach to solving time-dependent IPDAE systems by discretizing in space, but leaving the time variable continuous.

Neumann

Neumann is a type of boundary condition that specifies the values of the first-order spatial derivatives on the boundary.

6 Modeling Language Conventions

This section describes:

- Conventions used within the modeling language.
- Test conventions used within this manual when describing the modeling language.

Modeling Language Conventions

This section describes the conventions you must follow when writing modeling language.

Adding Comments

You can use two comment markers:

Symbol	Effect
//	Everything on the same line to the right of the // characters is commented out. A second // on the same line has no effect.
/* ... */	All text between /* and */ is commented out. /* and */ pairs can be nested.

In the model editor, all commented out text appears highlighted in green. However, note that the text editor does not highlight commented information after a nested comment.

Comments Examples

The following examples show the use of the comment markers:

```
// This is a comment


// This is a comment // This is a comment also
```

```
/*All text in this multi-line sentence is
commented out */
```

General Modeling Language Conventions

The modeling language is free format. This means that you can write syntax over several lines and insert spaces for clarity.

The modeling language is not case sensitive.

Text Conventions

The conventions used in the documentation that describes the modeling language are:

Convention	Example	Meaning
Text in courier font	FlowIn = FlowOut ;	Examples of modeling language.
...	Item , Item ...	A list of arguments.
:	MODEL HeatedTray USES Heater : END	More text is required but not shown.
Text in bold, upper case	**MODEL, CONNECT, AND**	Aspen Custom Modeler keywords.
Text in lower case with initial capital letters	VariableName, Value	You must supply a name, quantity, etc.
Italic text	VALUE:Value , *HIBOUND:HiBound* ;	This input is optional.

General Information

Copyright

Version Number: 2004.1
April 2005

Aspen ACOL™, Aspen Adsim®, Aspen Advisor™, Aspen Aerotran®, Aspen Alarm & Event™, Aspen APLE™, Aspen Apollo Desktop™, Aspen Apollo Online™, Aspen AssetBuilder™, Aspen ATOMS™, Aspen Automated Stock Replenishment™, Aspen Batch Plus®, Aspen Batch.21™, Aspen BatchCAD™, Aspen BatchSep™, Aspen Calc™, Aspen Capable-to-Promise®, Aspen CatRef®, Aspen Chromatography®, Aspen Cim-IO for ACS™, Aspen Cim-IO for Csi VXL™, Aspen Cim-IO for Dow MIF™, Aspen Cim-IO for G2™, Aspen Cim-IO for GSE D/3™, Aspen Cim-IO for Hewlett-Packard RTAP™, Aspen Cim-IO for Hitachi PLC (H04E)™, Aspen Cim-IO for Intellution Fix™, Aspen Cim-IO for Melsec™, Aspen Cim-IO for WonderWare InTouch™, Aspen Cim-IO for Yokogawa Centum CS™, Aspen Cim-IO for Yokogawa Centum XL™, Aspen Cim-IO for Yokogawa EW3™, Aspen Cim-IO Interfaces™, Aspen Cim-IO Monitor™, Aspen Cim-IO™, Aspen Collaborative Demand Management™, Aspen Collaborative Forecasting™, Aspen Compliance.21™, Aspen COMThermo TRC Database™, Aspen COMThermo®, Aspen Cost Factor Manual™, Aspen Crude Manager™, Aspen Crude Margin Evaluation™, Aspen Custom Modeler®, Aspen Data Source Architecture™, Aspen Decision Analyzer™, Aspen Demand Manager™, Aspen DISTIL™, Aspen Distribution Scheduler™, Aspen DMCplus® Composite, Aspen DMCplus® Desktop, Aspen DMCplus® Online, Aspen DPO™, Aspen Dynamics®, Aspen eBRS™, Aspen Enterprise Model™, Aspen ERP Connect™, Aspen FCC®, Aspen FIHR™, Aspen FLARENET™, Aspen Fleet Operations Management™, Aspen Framework™, Aspen FRAN™, Aspen Fuel Gas Optimizer Desktop™, Aspen Fuel Gas Optimizer Online™, Aspen General Construction Standards™, Aspen Hetran®, Aspen HX-Net®, Aspen Hydrocracker®, Aspen Hydrotreater™, Aspen HYSYS Amines™, Aspen HYSYS Crude™, Aspen HYSYS Dynamics™, Aspen HYSYS OLGAS 3-Phase™, Aspen HYSYS OLGAS™, Aspen HYSYS OLI Interface™,

Corporate

Aspen Technology, Inc.	Phone: (1) (617) 949-1000
Ten Canal Park	Toll Free: (1) (888) 996-7001
Cambridge, MA 02141-2201	Fax: (1) (617) 949-1030

USA URL: http://www.aspentech.com

Related Documentation

In addition to this document, the following documents are provided to help users learn and use the Aspen Utilities applications.

Title	Content
Aspen Custom Modeler 2004.1 Getting Started Guide	Contains basic hands-on tutorials to help you become familiar with Aspen Custom Modeler.
Aspen Custom Modeler 2004.1 User Guide	Contains a general overview of ACM functionality and more complex and extensive examples of using Aspen Custom Modeler.
Aspen Custom Modeler 2004.1 Library Reference	Contains reference information on control models, property procedure types, utility routines, port types, and variable types.
Aspen Custom Modeler 2004.1 Aspen Modeler Reference	Contains information on using automation, solver options, physical properties, the Control Design Interface (CDI)), Simulation Access eXtensions, online links, and using external nonlinear algebraic solvers.
Aspen Custom Modeler 2004.1 DMCplus® Controllers Interface	Contains information on using DMCplus with Aspen Custom Modeler or Aspen Dynamics™.
Aspen Custom Modeler 2004.1 Polymer Simulations with Polymers Plus	Polymers Plus is a layered product of Aspen Custom Modeler. It provides additional functionality to the properties package, Properties Plus, enabling polymers to be fully characterized in Aspen Custom Modeler models.

Technical Support

Online Technical Support Center

AspenTech customers with a valid license and software maintenance agreement can register to access the Online Technical Support Center at:

http://support.aspentech.com

You use the Online Technical Support Center to:

- Access current product documentation.
- Search for technical tips, solutions, and frequently asked questions (FAQs).
- Search for and download application examples.
- Search for and download service packs and product updates.
- Submit and track technical issues.
- Search for and review known limitations.
- Send suggestions.

Registered users can also subscribe to our Technical Support e-Bulletins. These e-Bulletins proactively alert you to important technical support information such as:

- Technical advisories.
- Product updates.
- Service Pack announcements.
- Product release announcements.

Phone and E-mail

Customer support is also available by phone, fax, and e-mail for customers who have a current support contract for their product(s). Toll-free charges are listed where available; otherwise local and international rates apply.

For the most up-to-date phone listings, please see the Online Technical Support Center at:

http://support.aspentech.com

Support Centers	Operating Hours
North America	8:00 – 20:00 Eastern time
South America	9:00 – 17:00 Local time
Europe	8:30 – 18:00 Central European time
Asia and Pacific Region	9:00 – 17:30 Local time

Index

M

N

O

P

Q

R

S

T

U

V

W