BSR/ASHRAE Standard 231P

Public Review Draft

A Control Description Language for Building Environmental Control Sequences

Second Public Review Draft (June 2025)
(Complete Draft for Full Review)

ASHRAE, 180 Technology Parkway, Peachtree Corners GA 30092

TABLE OF CONTENTS

(This foreword is not part of this standard. It is merely informative and does not contain requirements necessary for conformance to the standard. It has not been processed according to the ANSI requirements for a standard and may contain material that has not been subject to public review or a consensus process. Unresolved objectors on informative material are not offered the right to appeal at ASHRAE or ANSI.)

Foreword

The goal of this standard is to define and document a standard interchange format for the control logic to be used in control systems. Note that this standard is complementary and tightly connected to other work being developed in ASHRAE controls standards and guidelines. For example, ASHRAE Guideline 36 defines best practices for high performance sequences, while Guideline 13 defines how to develop a control specification. ASHRAE Standard 135 (BACnet) defines a protocol for controls communication, while proposed Standard 223P defines a methodology for semantic modeling. The intent is that this standard will be coordinated to work with the other ASHRAE standards and guidelines. Note that there is associated project work being done by the US Department of Energy's National Laboratories that is developing tools and control sequence libraries based on this standard. See the appendix B for more details of these efforts.

CDL is expected to be applied to these applications:

- *Guideline 36: Sequences of operation (SOOs) in Guideline 36 are currently written only in English. This format is mandatory for human understanding of the SOOs, but the language is inherently imprecise and ambiguous. This has led to inconsistent implementation of the SOOs. Writing the SOOs in CDL will eliminate these issues.*
- *Simulation. CDL can be used to simulate control sequences using Spawn of EnergyPlus (SOEP), or any other tool that supports the open Modelica standard, allowing designers to determine the energy efficiency effectiveness and to fine tune the logic. This could also be used to demonstrate the effectiveness of Guideline 36 SOOs for a given application.*
- *Control System Manufacturer Programming. Manufacturers of digital control systems could create translators that convert CDL to their proprietary programming language, allowing fast and reliable ways to convert CDL developed by and for Guideline 36 and energy modelers referenced above. Eventually some manufacturers may change from their proprietary programming language to CDL, just as they did with BACnet.*

1 PURPOSE

The purpose of this standard is to define a declarative graphical programming language for building environmental control sequences that are both human- and machine-readable designed for specification, implementation through machine-to-machine translation, documentation, and simulation.

2 SCOPE

This standard applies to building automation systems controlling environmental systems such as mechanical systems, active facades, and lighting.

3 DEFINITIONS

3.1 Definitions

Term	Definition
boolean	a data type with two possible values (true/false, which can be interpreted as open/close, or on/off).
composite block	a collection of any number of elementary blocks and other composite blocks, inputs, outputs, and connectors. *Informative note: Composite blocks are a valuable tool to allow for consistent re-use of logic*.
Control Description Language (CDL)	an interchange format for control sequence logic using a subset of the Modelica modeling language intended to be used for control sequence definition and allows for use in modeling and simulation as well as for use in commercially available control systems, or to be translated to CXF. *Informative note: See Section 5.0 for more details.*
constant	value that does not change.
control logic	functional description which determines how outputs are assigned based on given inputs, parameters, and state variables. Control logic is the digital implementation of what is specified in the control sequence.
control sequence	English language description that is used to specify the control logic. Control sequences are typically developed by HVAC / Control system designers.
Control Exchange Format (CXF)	CDL encoded in a JSON-LD format intended to be used as a syntax to represent CDL information for import by control system providers using translators; and to exchange control logic between control systems. *Informative Note: See Section 6.0 for more details.*
elementary block	mathematical function, including specification of the data types of its inputs, outputs, and parameters. *Informative Note: See Section 7.0 for more details.*
extension block	elementary blocks beyond those defined in this standard, including inputs, outputs, and parameters, defined in a manner that is interoperable with this standard. *Informative Note: See Section 5.0 for more details*

integer	refers to a data type of whole values (e.g.-1, 0, 1, 2….)
Java Script Object Notation (JSON)	a standard text-based data exchange format. *Informative note: See json.org.*
Modelica	a modeling language governed by the Modelica Association. *Informative note: See modelica.org.*
parameter	a variable whose value is not changed dynamically by the control logic.
proprietary function (system functions)	logic that provides functions not defined in this standard, such as optimal start, scheduling, and alarming.
real	a data type of continuous values.
simulation	a virtual representation of a real-world system.

4 HOW TO USE THIS DOCUMENT

This document consists of three primary technical sections. These include:

- Section 5: Control Description Language (CDL). This section contains all the definitions of CDL which is a subset of the Modelica modeling language.
- Section 6: Control eXchange Format (CXF). This section has the details of how the JSON definition for control logic is expressed.
- Section 7: Elementary Blocks. This contains all the definitions for the elementary function blocks defined in the standard.

More details about the concept and structure of CDL and CXF can be found in Informative Appendix A.

5 CONTROL DESCRIPTION LANGUAGE

CDL defines the syntax for implementing the logic and documentation of a control logic. CDL is used for the control logic design, as well as for control logic simulation. Commercially available control system tools used for control logic programming shall be able to import and export logic using either CDL or CXF.

Informative note: CDL is defined using a subset of a modeling language called Modelica. The use of Modelica allows the use of open source and commercially available tools that can be utilized to simulate both the control logic (as represented in CDL) as well as the corresponding mechanical system (represented in Modelica).

5.1 Basic Elements of CDL

CDL consists of the following elements:

- Elementary Blocks (defined in Section 7.0 of this standard).

- Connectors through which these blocks receive input values and make accessible output values.
- Permissible data types.
- Syntax to specify:
    - how to instantiate blocks and assign values to parameters.
    - how to connect inputs of blocks to outputs of other blocks.
    - how to document blocks.
    - how to add annotations.
    - how a group of blocks and connectors can be used to define a Composite Block.
    - how to support new (extension) blocks that are not part of this standard.
- A model of computation that describes when blocks are executed and when outputs are assigned to inputs.

5.2 Syntax

Informative note: CDL is defined using a subset of the modeling language Modelica. The use of Modelica allows users to view, modify, and simulate CDL-conformant control logic with any Modelica-compliant tool. One constraint with the use of Modelica is that certain terms are defined or reserved.

The following Modelica keywords are not supported in CDL:

- inner and outer for instance hierarchy lookup
- break for component deselection

The following Modelica language features are not supported in CDL:

- clocks for clocked state machine
- algorithm sections to express sequences of statements
- initial equation and initial algorithm sections for system initialization

5.3 Units

CDL shall use a specific set of SI units as designated in the table below. If necessary, tools shall convert these units to display user-selected units:

Table 5-1: Units in CDL

Measurement	**CDL units**
Area	m2
Electric or Thermal Energy Use	J
Flow Mass	kg/s
Flow Volume	m3/s
Heat Flow Rate (boiler, chiller, etc.)	W
Power (chillers, fans, pumps, etc.)	W
Pressure (air, water, etc.)	Pa
Relative Humidity	Ratio between 0 and 1

Specific Enthalpy	J/kg
Static Pressure (water)	Pa
Temperature	K
Time	s
Velocity	m/s

5.4 Permissible Data Types

5.4.1 Data Types

Informative note: CDL is limited to a specific set of supported data types. Note that CXF has a broader range of data types that are supported. The data types in CDL utilize a subset of the parameters defined in Modelica. Only the parameters that are translated to CXF are included in this standard.

5.4.1.1 Real Type

The following defines the Real type:

```
Type Real // Note: Defined with Modelica syntax although predefined
  RealType value; // Accessed without dot-notation
  parameter StringType quantity    = "";
  parameter StringType unit       = "" "Unit used in equations";
  parameter StringType displayUnit = "" "Default display unit";
  parameter RealType min=-Inf, max=+Inf; // Inf denotes a large Value
  parameter RealType nominal = 1;  // Nominal value, for error control
  parameter BooleanType unbounded = false; // For error control
```

Real Type/double shall follow IEC 60559:1989 (ANSI/IEEE 754-1985) double format.

Attribute Descriptions:

- `quantity`: The quantity attribute is optional, can take on the following values:
  - "", which is the default, is considered as no quantity being specified.
  - `Angle` for area (such as used for sun position).
  - `Area` for area.
  - `Energy` for energy.
  - `Frequency` for frequency.
  - `Illuminance` for illuminance.
  - `Irradiance` for solar irradiance.
  - `MassFlowRate` for mass flow rate.
  - `MassFraction` for mass fraction.
  - `Power` for power.
  - `PowerFactor` for power factor.
  - `Pressure` for absolute pressure.
  - `PressureDifference` for pressure difference.

  - `SpecificEnergy` for specific energy.
  - `TemperatureDifference` for temperature difference.
  - `Time` for time.
  - `ThermodynamicTemperature` for absolute temperature.
  - `Velocity` for velocity.
  - `VolumeFlowRate` for volume flow rate.
  - `Current` for electrical current (such as the chiller demand).
  - `ThermalRampRateTime` for thermal ramp rate by time.
  - `ThermalRampRateTemperature` for thermal ramp rate by temperature.

- `unit`: The units for the data value that are utilized in CDL. In CDL, engineering units are pre-defined to follow SI standards – see Section 5.3 for details.

- `displayUnit`: It is used to show the localized value for how to display units to the user. *Informative note: It is up to the user to specify which unit to be shown. As an example, for a project in the US, temperatures are often shown in degrees F, so a displayUnit value of `degF` would be used. Tools that implement CDL may convert the value from unit to displayUnit before showing it in a GUI or a log file.*

- `nominal`: This attribute is used for scaling purposes and to define tolerances.

- `unbounded`: This attribute is used by solvers for integrating quantities that can grow over multiple orders of magnitude.

5.4.1.2 Integer Type

The following defines the Integer type:

```
type Integer // Note: Defined with Modelica syntax although predefined
  IntegerType value; // Accessed without dot-notation
  parameter IntegerType min=-Inf, max=+Inf;
end Integer;
```

The minimal number range for `IntegerType` shall be from -2147483648 to +2147483647, corresponding to a two's-complement 32-bit integer implementation.

The Integer data type shall not include units.

5.4.1.3 Boolean Type

The following defines the Boolean type:

```
type Boolean // Note: Defined with Modelica syntax although predefined
  BooleanType value; // Accessed without dot-notation
end Boolean;
```

5.4.1.4 String Type

The following defines the String type:

```
type String // Note: Defined with Modelica syntax although predefined
  StringType value; // Accessed without dot-notation
end String;
```

5.4.1.5 Enumeration Type

A declaration of the form

```
type E = enumeration([enumList]);
```

defines an enumeration type `E` and the associated enumeration literals of the `enumList`. The enumeration literals shall be distinct within the enumeration type. The names of the enumeration literals are defined inside the scope of `E`. Each enumeration literal in the `enumList` has type `E`.

Informative note: For example:

```
type SimpleController = enumeration(P, PI, PD, PID);
parameter SimpleController = SimpleController.P;
```

An optional comment string can be specified with each enumeration literal.

Informative note: For example:

```
type SimpleController = enumeration(
  P "P controller",
  PI "PI controller",
  PD "PD controller",
  PID "PID controller")
"Enumeration defining P, PI, PD, or PID simple controller type";
```

5.4.2 Parameter and Constant Declarations

Informative note: A parameter is an exposed value that is used to configure an Elementary, Composite, or Extension Block. These values are set by the user when setting up the sequence. The value of a parameter cannot be changed through an input connector. Parameters are values that do not depend on time; however, their values can be changed during run-time through a user interaction with the control program (such as to change a control gain), unless a parameter is a structural parameter. A constant is a value that is fixed at compilation time.

Parameters shall be declared with the `parameter` prefix.

Informative note: For example, to declare a proportional gain, use

```
parameter Real k(min=0) = 1 "Proportional gain of controller";
```

Constants are declared with the `constant` prefix.

Informative note: For example,

```
constant Real pi = 3.14159;
```

To avoid that the value of a `parameter` or `constant` may be changed when instantiating a block, the `final` keyword shall be prepended to the `parameter` or `constant` declaration.

Informative note: For example, in Composite Blocks, a library implementer may want to avoid that the user changes certain parameter values.

5.4.3 Arrays

Array indices shall be of type Integer only. The first element of an array has index 1. An array of size 0 is an empty array.

Values of arrays shall be declared using one of the approaches below:

- the notation `{x1, x2, ...}`
- one or several iterators
- a `fill` or `cat` function

Informative note: Arrays may be used, for example, in the creation of libraries so that the logic is applicable to any number of chillers. Not all control systems support arrays, and it is possible to flatten them when rendered in CXF. Each of these data types, including Elementary Blocks, Composite Blocks, Extension Blocks, and Connectors can be a single instance, one-dimensional array or n-dimensional array (matrix).

For example, the following declarations all assign the array `{1, 2, 3}` to parameters:

```
parameter Real k1[3] = {1, 2, 3};
parameter Real k2[3] = {i for i in 1:3};
parameter Real k3[3] = k1;
parameter Real k4[3] = fill(1, 3) + {0, 1, 2};
parameter Real k5[3] = cat(1, {1}, {2}, {3});
```

The following declaration instantiates two blocks and sets the value of the parameter k to 2 and 3:

```
MultiplyByParameter mul[2](k={2, 3});
```

Notes::

- *The use of arrays is beneficial in defining a control logic if it acts on any number of equipment (such as on any number of chillers, and the number is only known when the sequence is used for a specific project).*
- *Many control systems do not support arrays.*
- *The size of arrays will be fixed at translation. It cannot be changed during run-time.*
- *Enumeration or Boolean data types are not permitted as array indices.*

5.5 Encapsulation of Functionality

All computations shall be encapsulated in a block. Blocks expose parameters for configuring the block and expose inputs and outputs using *connectors*.

Informative note: Blocks are either Elementary Blocks, Composite Blocks, or Extension Blocks.

5.6 Elementary Blocks

Section 7.0 defines a set of Elementary Blocks. Elementary blocks must be used as defined and cannot be modified.

Informative note: Elementary Blocks along with inputs, outputs, and connectors, and along with other Composite or Extension Blocks, are used to compose control logic.

For example: A tool that is CDL compliant and implements the elementary function AddParameter may implement it as:

```
block AddParameter "Output the sum of an input plus a parameter"
  parameter Real p "Value to be added";
  CDL.Interfaces.RealInput u "Connector of Real input signal";
  CDL.Interfaces.RealOutput y "Connector of Real output signal";
equation
  y = u + p;
annotation(Documentation(info("
  <html>
  <p>
  Block that outputs <code>y = u + p</code>,
  where <code>p</code> is parameter and <code>u</code> is an input.
  </p>
  </html>"));
end AddParameter;
```

5.7 Connectors

Elementary Blocks, Composite Blocks, and Extension Blocks expose their inputs and outputs through input and output connectors.

The permissible connectors are stored in the CDL library in the package `CDL.Interfaces` and they are `BooleanInput`, `BooleanOutput`, `IntegerInput`, `IntegerOutput`, `RealInput`, and `RealOutput`. Connectors do not carry Enumeration and String data.

Connectors must not be in a `protected` section.

Connectors carry scalar variables, vectors, or arrays of values each of which has the same data type. For arrays, the connectors must be explicitly declared as an array.

Informative note: For example, to declare an array of `nin` input signals, use

```
parameter Integer nin(min=1) "Number of inputs";
CDL.Interfaces.RealInput u[nin] "Connector for 2 Real input signals";
```

Many building control product tools only support scalar variables on graphical connections. This leads to the situation that different control logics need to be implemented for any combination of equipment. For example, if only scalars are allowed in connections, then a chiller plant with two chillers needs a different sequence than a chiller plant with three chillers. With vectors, however, one sequence can be implemented for chiller plants with any number of chillers.

If control product tools do not support vectors on connections, then during translation from CDL to CXF, the vectors (or arrays) can be flattened, e.g., the arrays are converted to scalars. For example, blocks of the form

```
    parameter Integer n = 2 "Number of blocks";
    CDL.Reals.Sources.Constant con[n](k={1, 2});
    CDL.Reals.MultiSum mulSum(nin=n); // multiSum that contains an input con-
nector u[nin]
  equation
    connect(con.y, mulSum.u);
```

could be translated to the equivalent of

```
    CDL.Reals.Sources.Constant con_1(k=1);
    CDL.Reals.Sources.Constant con_2(k=1);
    CDL.Reals.MultiSum mulSum(nin=2);
  equation
    connect(con_1.y, mulSum.u_1);
    connect(con_2.y, mulSum.u_2);
```

E.g., two instances of `CDL.Reals.Sources.Constant` *are used, the vectorized input* `mulSum.u[2]` *is flattened to two inputs, and two separate connections are instantiated. This will preserve the control logic, but the components will need to be graphically rearranged after translation.*

5.8 Composite Blocks

CDL defines composition rules that instantiate parameters, inputs, outputs, and other blocks, and saves them as a Composite Block.

Informative note: A Composite Block could represent all the control logic used to control a device; hence the Composite Block could be translated to a control product line; or it could be instantiated in another Composite Block to compose a hierarchical control logic for a device. For example, a control logic for an air handler unit may be a Composite Block that contains instances of Composite Blocks for the freeze protection, for the heating coil, or for the cooling coil.

Figure 5-1 shows a simple composite block.

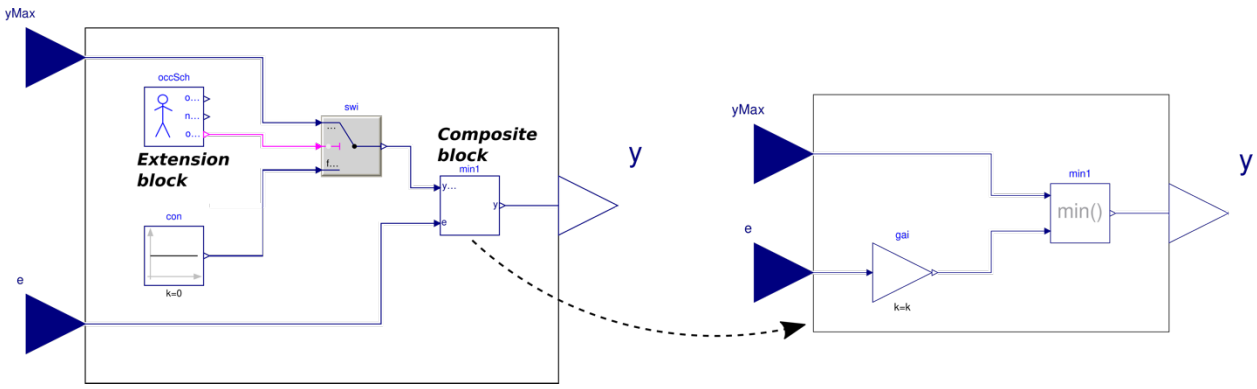


Figure 5-1 Example of a composite control block that includes elementary blocks, an extension block (Section 5.9), and a sub-composite block.

5.8.1 Implementation of Composite Blocks

Composite Blocks are declared using the syntax:

```
within PackageName;
block BlockName "Description"
 …
 end BlockName;
```

`PackageName` is the name of the package in which the block is stored. The `BlockName` is the user-selected name of the block. The "`Description`" is a one-line string. The three dots … contains declarations that instantiate constants, parameters, Elementary Blocks, Composite Blocks, and Extension Blocks followed by an `equation` section that contains the connect statements and the `annotation` that provides the documentation of the Composite Block.

The Composite Block must be stored on the file system under the name of the Composite Block with the file extension `.mo`, and with each package name being a directory. The name must be an allowed Modelica class name (see https://specification.modelica.org/master/class-predefined-types-and-declarations.html).

Informative note:
For example, if a user specifies a new Composite Block MyController.MyAdder, then it shall be implemented as

```
within MyController;
block MyAdder "Description"
…
end MyAdder;
```

stored in the file MyController/MyAdder.mo on Linux or OS X, or MyController\MyAdder.mo on Windows.

Composite Blocks may contain tens or hundreds of instances of other Composite Block, some of them may be conducting such low-level operation that are of no interest to someone who uses the Composite Block. Therefore, Composite Blocks allow instances of Elementary Blocks, Composite Blocks, and Extension Blocks to be declared after a keyword called `protected`*. If a simulation program or a building automation system stores inputs and outputs of blocks, or makes their value readable via a browser, then these protected instances should by default not be shown. The protected keyword is often used to avoid clutter in such browser that would distract the user and may make it more time consuming to find relevant data; or that may lead to large output files or longer computing times for no useful reason.*

5.8.2 Equations

In Composite Blocks, after all instantiations, a keyword `equation` must be present to introduce the equation section. The equation section can only contain connections and annotations. Unlike in Modelica, an `equation` section shall not contain equations such as `y=2*u;` or commands such as `if`, `while` and `when`.

There shall not be an initial equation, initial algorithm, or algorithm section.

5.8.3 Assigning of Values to Parameters

Informative note: Parameters are values used to configure Elementary Blocks, Composite Blocks, and Extension Blocks. The values of parameters can be changed through a user interaction with the control program (such as to change a control gain) unless a parameter is a structural parameter such as the size of an array, or unless a parameter is declared as `final`. *Optionally, it is also possible to assign the return value of a function to a parameter. For example, this can be used when creating a library of sequences.*

The declaration of parameters and their values follows conventions for Modelica but is limited to the type of expressions that are allowed in such assignments. For Boolean parameters, only expressions involving `and`, `or`, and `not` and the functions in Table 5.1 are allowed. For Real and Integer, expressions are allowed that involve:

- the basic arithmetic functions +, -, *, /,
- the relations >, >=, <, <=, ==, <>,
- calls to the functions listed in Table 5.1.

Table 5-2: Functions that are allowed in parameter assignments

Function	Description
`abs(v)`	Returns the absolute value of `v`.
`sign(v)`	Returns if `v>0` then `1` else if `v<0` then `-1` else `0`.
`sqrt(v)`	Returns the square root of `v` if `v>=0`, or an error otherwise.
`div(x, y)`	Returns the algebraic quotient `x/y` with any fractional part discarded (also known as truncation toward zero). [Note: this is defined for / in C99; in C89 the result for negative numbers is implementation-defined, so the standard function `div()` must be used.]. Result and arguments must have type `Real` or `Integer`. If either of the arguments is `Real`, the result is `Real`; otherwise, it is `Integer`.
`mod(x, y)`	Returns the integer modulus of `x/y`, i.e., `mod(x,y)=x-floor(x/y)*y`. Result and arguments shall have type `Real` or `Integer`. If either of the arguments is `Real`, the result is `Real`; otherwise, it is `Integer`. *Informative note: Examples are* `mod(3,1.4)=0.2`, `mod(-3,1.4)=1.2` *and* `mod(3,-1.4)=-1.2`.
`rem(x,y)`	Returns the integer remainder of `x/y`, such that `div(x,y)*y + rem(x, y) = x`. Result and arguments shall have type `Real` or `Integer`. If either of the arguments is `Real`, the result is `Real`; otherwise, it is `Integer`. *Informative note: Examples are* `rem(3,1.4)=0.2` *and* `rem(-3,1.4)=-0.2`.
`ceil(x)`	Returns the smallest integer not less than `x`. Result and argument shall have type `Real`.
`floor(x)`	Returns the largest integer not greater than `x`. Result and argument shall have type `Real`.
`integer(x)`	Returns the largest integer not greater than `x`. The argument shall have type `Real`. The result has type `Integer`.
`min(A)`	Returns the least element of array expression `A`.

`min(x, y)`	Returns the least element of the scalars `x` and `y`.
`max(A)`	Returns the greatest element of array expression `A`.
`max(x, y)`	Returns the greatest element of the scalars `x` and `y`.
`sum(...)`	The expression `sum(e(i, ..., j) for i in u, ..., j in v)` returns the sum of the expression `e(i, ..., j)` evaluated for all combinations of `i` in `u`, `...`, `j` in `v`: `e(u[1], ... ,v[1]) + e(u[2], ... ,v[1])+... +e(u[end],... ,v[1])+...+e(u[end],... ,v[end])`. The type of `sum(e(i, ..., j) for i in u, ..., j in v)` is the same as the type of `e(i,...j)`.
`fill(s, n1, n2, ...)`	Returns the $n_1 \times n_2 \times n_3 \times \ldots$ array with all elements equal to scalar or array expression `s` $(n_i \geq 0)$. The returned array has the same type as `s`. Recursive definition: `fill(s, n1, n2, n3, ...) = fill( fill(s, n2, n3, ...), n1);, fill(s,n)={s, s, ..., s}`. The function needs two or more arguments; that is `fill(s)` is not legal.
`size(...)`	Returns dimensions of an array. For $1ei \leq n$, where n is the number of dimensions in `A`, the expression `size(A,i)` returns the size of dimension i of array expression `A`. The expression `size(A)` returns a vector of length n containing the dimension sizes of `A`. *Informative note: Examples are* `size([1, 2, 3; 3, -4, 5], 1)=2` *and* `size([1, 2, 3; 3, -4, 5])={2,3}`.
`cat(k, A, B, C, …)`	Returns an array that concatenates arrays `A, B, C, …` along dimension `k`, according to the following rules: • Arrays `A, B, C, …` must have the same number of dimensions, • Arrays `A, B, C, …` must be type compatible expressions giving the type of the elements of the results. The maximally expanded types should be equivalent. `Real` and `Integer` subtypes can be mixed resulting in a `Real` result array where the `Integer` numbers have been transformed to `Real` numbers. • `k` has to characterize an existing dimension; `k` shall be an integer number. • Size matching: Arrays `A, B, C, ...` must have identical array sizes with the exception of the size of dimension `k`.

Informative note: For example, the following could be used to instantiate a gain and assign the value −1 or 2 to its parameter k:

```
CDL.Gain gai(k=-1) "Constant gain of -1" annotation(...);
CDL.Gain doubleTheInput(final k=2) annotation(...);
```

where the documentation string is optional. Here we used the `final` keyword to avoid a user changing the value of `k`. The annotation is typically used for the graphical positioning of the instance in a block diagram.

Using expressions in parameter assignments and propagating values of parameters in a hierarchical formulation of control logic are convenient language constructs to express relations between parameters and are invaluable in defining libraries of sequences. However, many commercial control product lines do not support arrays or the propagation of parameter values and evaluation of expressions in parameter assignments.

To deal with this limitation, CDL can be represented for a specific instance that does not include the propagated parameters and that evaluate expressions that involve parameters. See Figure 2 for the relationship between a library of sequences and a specific instance of a sequence.

5.8.4 Conditionally Removing Instances

CDL supports conditionally removing instances of blocks, inputs, and outputs and their connections. When an instance needs to be conditionally removed, an `if` clause must be used.

Informative note: For example, to have an implementation of a sequence that optionally includes code to use an occupancy sensor, when available, or to remove the code when it is not provided. An example code snippet is

```
    parameter Boolean have_occSen=false
      "Set to true if zones have occupancy sensor";
    CDL.Interfaces.IntegerInput nOcc if have_occSen
      "Number of occupants"
      annotation (__cdl(default = 0));
    CDL.Reals.MultiplyByParameter gai(k = VOutPerPer_flow)
      if have_occSen
      "Outdoor air per person";
  equation
    connect(nOcc, gai.u);
```

By the Modelica language definition, all connections to `nOcc` *will be removed if* `have_occSen = false`.

Many control product tools may not support conditionally removing instances. Rather, these instances are always present, and a value for the input must be present. To accommodate this, every input connector that can be conditionally removed can declare a default value of the form `__cdl(default = value)`*, where value is the default value that will be used if the building automation system does not support conditionally removing instances. The type of value must be the same as the type of the connector. For Boolean connectors, the allowed values are* `true` *and* `false`.
If the `__cdl(default = value)` *annotation is absent, then the following values are assumed as default:*

- *For* `RealInput`*, the default values are:*
  - *If* `unit=K`*: If* `quantity="TemperatureDifference"`*, the default is* 0 `K`*, otherwise it is* `293.15 K`.
  - *If* `unit=Pa`*: If* `quantity="PressureDifference"`*, the default is* 0 `Pa`*, otherwise it is* `101325 Pa`.
  - *For all other units, the default value is* 0.
- *For* `IntegerInput`*, the default value is* 0.

- *For* `BooleanInput`*, the default value is* `false`*.*

Note that output connectors must not have a specification of a default value because if a building automation system cannot conditionally remove instances, then the block (or input connector) upstream of the output will always be present (or will have a default value).

5.8.5 Points lists

Points lists documentation is supported within a CDL-conforming sequences. This is an optional function. Tools do not have to support it. When it is utilized, it must not cause errors in the tool.

For point lists,

- the connectors `RealInput` and `IntegerInput` are analog inputs if translated to CXF.
- the connectors `RealOutput` and `IntegerOutput` are analog outputs if translated to CXF.
- the connectors `BooleanInput` and `BooleanOutput` are digital inputs and outputs.

5.8.5.1 Annotations that Cause Point Lists to be Generated

To enable generation of point lists, annotations that are written as `__cdl(…)` must be used. The annotation

```
__cdl(generatePointlist=Boolean, controlledDevice=String);
```

at the class level specifies that a point list of the sequence is generated. If not specified, it is assumed that `__cdl(generatePointlist=false)`. The key `controlledDevice` is optional. It is used to list the device that is being controlled. Its value will be written to the point list, but not used otherwise.
When instantiating an Elementary Block, a Composite Block, or an Extension Block, with the annotation `__cdl(generatePointlist=Boolean)` being added to the instantiation clause, the annotation will override the class level declaration.

Informative note: For example,

```
block A
  MyController con1;
  MyController con2 annotation(__cdl(generatePointlist=false));
  annotation(__cdl(generatePointlist=true));
end A;
```

generates a point list for `A.con1` *only, while*

```
block A
  MyController con1;
  MyController con2 annotation(__cdl(generatePointlist=true));
  annotation(__cdl(generatePointlist=false));
end A;
```

generates a point list for `A.con2` *only.*

The `generatePointlist` annotation will be propagated down in an Elementary Block, a Composite Block, or an Extension Block by specifying in the instantiation clause the annotation

```
__cdl(propagate(instance=String, generatePointlist=true));
```

Controllers deeper in the hierarchy are referred to using the dot notation.

Informative note: For example, in `instance="subCon1.subSubCon1"` *where* `subSubCon1` *is an instance of an Elementary or a Composite Block in* `subCon1`.

The value of `instance` must be the name of an Elementary Block, a Composite Block, or an Extension Block. It must be declared. When the value is a conditionally removable block and is removed, the declaration will be safely ignored. Higher-level declarations override lower-level declarations.

Informative note: For example, assume `con1` *has a block called* `subCon1`. *Then, the declaration*
```
MyController con1 annotation(__cdl(propagate(instance="subCon1", generate-
Pointlist=true)));
```
sets `generatePointlist=true` *in the instance* `con1.subCon1`.

It allows any number of `propagate(...)` annotations for a controller.

Informative note: Specifying multiple `propagate(...)` *annotations is useful for composite controllers. For example,*
```
MyController con1 annotation(
  __cdl(
    propagate(instance="subCon1", generatePointlist=true),
    propagate(instance="subCon1.subSubCon1", generatePointlist=true),
    propagate(instance="subCon1.subSubCon2", generatePointlist=false)
  )
);
```
allows a fine-grained propagation to individual blocks of a Composite Block.

5.8.5.2 Annotations for Connectors

For generating point lists, an annotation of the form for the Connectors
```
__cdl(connection(hardwired=Boolean));
```
specifies whether the connection is hardwired or not. If the annotation is not presented, the Connector is not hardwired. The field `hardwired` has default value `false`.

An annotation of the form for the Connectors
```
__cdl(trend(interval=Real, enable=Boolean));
```
specifies if the Connector value is recommended to be trended and the interval to trend the value. The field `interval` must be specified, and its value is the trending interval in seconds. The field `enable` is optional, with default value of `true`, and it can be used to overwrite the value used in the sequence declaration.

The `connection` annotation will be propagated down in an Elementary Block, a Composite Block, or an Extension Block by specifying in the instantiation clause the annotation
```
__cdl(propagate(instance=String, connection(hardwired=Boolean)));
```

The `trend` annotation will be propagated down in an Elementary Block, a Composite Block or an Extension Block by specifying in the instantiation clause the annotation
```
__cdl(propagate(instance=String, trend(interval=Real, enable=Boolean)));
```

To propagate both the `connection` and the `trend` annotations together for one connector, specifying in the instantiation clause the annotation

```
__cdl(propagate(instance=String, connection(hardwired=Boolean), trend(inter-
val=Real, enable=Boolean)));
```

The value assigned to `instance` must be the instance name of a connector.

Informative note: If a Composite Block contains a Composite Block `con1`, which in turn contains a block `subCon1` that has an input `u`, the declaration

```
MyController con1 annotation(
  __cdl(propagate(instance="subCon1.u", connection(hardwired=Boolean)));
```

can be used to set the type of connection of input (or output) `con1.subCon1.u`.
Similarly, the declaration

```
MyController con1 annotation(
  __cdl(propagate(instance="subCon1.u", trend(interval=Real, enable=Bool-
ean)));
```

can be used to set how to trend that input (or output).
To combine the propagation, use the declaration

```
MyController con1 annotation(
  __cdl(propagate(instance="subCon1.u", connection(hardwired=Boolean),
trend(interval=Real, enable=Boolean)));
```

- The value assigned to `instance` must be the name of an instance that exist. If the instance is removable and it is removed, the annotation will be safely ignored.
- The higher-level declarations override lower-level declarations, and
- The annotation allows any number of `propagate(…)` annotations.

Informative note:
For example, consider the pseudo-code

```
block Controller
   Interfaces.RealInput u1
     annotation(__cdl(connection(hardwired=true), trend(interval=60, ena-
ble=true)));
   Interfaces.RealInput u2
     annotation(__cdl(connection(hardwired=false),
                      trend(interval=120, enable=true),
                      propagate(instance="con1.u1",
                        connection(hardwired=false),
                        trend(interval=120, enable=true))));

   MyController con1 annotation(__cdl(generatePointlist=true));
   MyController con2 annotation(__cdl(generatePointlist=false,
                                      propagate(instance="subCon1", generate-
Pointlist=true),
                                      propagate(instance="subCon2", generate-
Pointlist=true)));

equation
   connect(u1, con1.u1);
   connect(u2, con1.u2);
   connect(u1, con2.u1);
   connect(u2, con2.u2);
```

```
   annotation(__cdl(generatePointlist=true));
end Controller;

...

block MyController
   Interfaces.RealInput u1
     annotation(__cdl(connection(hardwired=false), trend(interval=120, ena-
ble=true)));
   Interfaces.RealInput u2
     annotation(__cdl(connection(hardwired=true),  trend(interval=60,  ena-
ble=true)));
   ...
   SubController1 subCon1;
   SubController2 subCon2;
   ...
   annotation(__cdl(generatePointlist=true));
end MyController;
```

A translator will generate an annotation propagation list as shown below. There will be a points list for `Controller`, `Controller.con1`, `Controller.con2.subCon1` *and* `Controller.con2.subCon1`. *Also, the annotation* `connection(hardwired=true), trend(interval=60, enable=true)` *of* `con1.u2` *will be overridden as* `connection(hardwired=false), trend(interval=120, enable=true)`.

```
[
  {
    "className": "Controller",
    "points": [
      {
        "name": "u1",
        "hardwired": true,
        "trend": {
          "enable": true,
          "interval": 60
        }
      },
      {
        "name": "u2",
        "hardwired": false,
        "trend": {
          "enable": true,
          "interval": 120
        }
      }
    ]
  },
  {
    "className": "Controller.con1",
    "points": [
      {
        "name": "u1",
        "hardwired": false,
        "trend": {
          "enable": true,
```

```
          "interval": 120
        }
      },
      {
        "name": "u2",
        "hardwired": false,
        "trend": {
          "enable": true,
          "interval": 120
        }
      }
    ]
  },
  {
    "className": "Controller.con2.subCon1",
    "points": [
      ...
    ]
  },
  {
    "className": "Controller.con2.subCon2",
    "points": [
      ...
    ]
  }
]
]
```

For an example of a point list generation, consider the pseudo-code shown below.

```
within Buildings.Controls.OBC.ASHRAE.G36_PR1.TerminalUnits
block Controller "Controller for room VAV box"
   ...;
   CDL.Interfaces.BooleanInput uWin "Windows status"
      annotation (__cdl(connection(hardwired=true),
                       trend(interval=60, enable=true)));
   CDL.Interfaces.RealOutput yVal "Signal for heating coil valve"
      annotation (__cdl(connection(hardwired=false),
                       trend(interval=60, enable=true)));
   ...
annotation (__cdl(generatePointlist=true, controlledDevice="Terminal unit"));
```

It specifies that a point list should be generated for the sequence that controls the system or equipment specified by `controlledDevice`*, that* `uWin` *is a digital input point that is hardwired, and that* `yVal` *is an analog output point that is not hardwired. Both can be trended with a time interval of 1 minute. The point list table will look as shown below.*

System/Equipment	Name	Type	Hardwired?	Trend [s]	Description
Terminal unit	`uWin`	DI	Yes	60	Windows status
Terminal unit	`yVal`	AO	No	60	Signal for heating coil valve
…	…	…	…	…	…

5.8.5.3 Control point properties

The control point has following properties:

- System/Equipment: It specifies which system/equipment the sequence is controlling.
- Name: It specifies the control point name.
- Type: It specifies the control point type. The types include DI (digital input), DO (digital output), AI (analog input), and AO (analog output).
- Hardwired: It specifies if the control point is hardwired. The value "Yes" means that the point is hardwired and the value "No" means that the point is not hardwired.
- Trend: It specifies the interval in seconds to trend the value. When the property is not specified and when the value equal 0, the point value will not be trended.
- Description: It describes what the control point variable is. If the control point is an output, it describes what the setpoint is; if the control point is an input, it describes what the measurement or setpoint is.

5.8.6 Connections

Connections connect input to output connectors. In other words, inputs and outputs are connectors that receive or make accessible values, whereas connections connect inputs to outputs, thereby ensuring that the value of an input is the same as the value of the connected output.

Informative note: In the graphical representation, the connectors are the triangles that render every input and output, and the connections are the lines that connect them.

For scalar connectors, each input connector of a block needs to be connected to exactly one output connector of a block. For vectorized connectors, or vectorized instances with scalar connectors, each or each element of an input connector needs to be connected to exactly one or one element of an output connector. Connections are listed after the instantiation of the blocks in an `equation` section. The syntax is `connect(port_a, port_b) annotation(...);` The `annotation(...)` declares the graphical rendering of the connection.

Informative note: The order of the connections and the order of the arguments in the connect statement does not matter. For example, to connect an input `u` of an instance `gain` to the output `y` of an instance `maxValue`, one would declare

```
CDL.Reals.Max maxValue "Output maximum value";
CDL.Reals.Gain gain(k=60) "Gain";

equation
  connect(gain.u, maxValue.y);
```

Only connectors that carry the same data type allow to be connected.
Attributes of the variables that are connected are handled as follows:

- If the `quantity`, `unit`, `min`, or `max` attributes are set to a non-default value for both connector variables, then they must be equal. ~~Otherwise, an error must be issued.~~

- If only one of the two connector variables declare the `quantity`, `unit`, `min`, `max`, `nominal`, or `unbounded` attribute, then this value is applied to both connector variables.
- If two connectors have different values for the `displayUnit` attribute, then either can be used.

Informative note: For example,

```
CDL.Reals.Max maxValue(y(unit="m/s")) "Output maximum value";
CDL.Reals.Gain gain(k=60) "Gain";
CDL.Reals.Gain gainOK(u(unit="m/s" ), k=60) "Gain";
CDL.Reals.Gain gainWrong(u(unit="kg/s"), k=60) "Gain";

equation
  connect(gain.u, maxValue.y); // This sets gain.u(unit="m/s") as gain.u does
                               // not declare its unit
  connect(gainOK.u, maxValue.y); // Correct, because unit attributes are
                                 // consistent
  connect(gainWrong.u, maxValue.y); // Not allowed, because of inconsistent
                                    // unit attributes
```

Signals shall be connected using a `connect` statement; assigning the value of a signal in the instantiation of the output connector is not allowed.

Informative note:
This ensures that all control logic is expressed as block diagrams. For example, the following model is valid:

```
block MyAdderValid
  Interfaces.RealInput u1;
  RealInput u2;
  Interfaces.RealOutput y;
  Continuous.Add add;
equation
  connect(add.u1, u1);
  connect(add.u2, u2);
  connect(add.y, y);
end MyAdderValid;
```

whereas the following implementation is not valid in CDL, although it is valid in Modelica

```
block MyAdderInvalid
  Interfaces.RealInput u1;
  Interfaces.RealInput u2;
  Interfaces.RealOutput y = u1 + u2; // not allowed
end MyAdderInvalid;
```

5.8.7 Annotations

Annotations must follow the same rules as described in the following Modelica 3.6 Specification:

- 18.2 Annotations for Documentation
- 18.6 Annotations for Graphical Objects, except for
  - 18.6.7 User input

- 18.8 Annotations for Version Handling

Informative note: For CDL, annotations are primarily used to graphically visualize block layouts, graphically visualize input and output signal connections, and to declare vendor annotations, (Section 18.4 in Modelica 3.6 Specification).

For CDL implementations of sources such as ASHRAE Guideline 36, any instance, such as a parameter, input, or output, that is not provided in the original documentation shall be annotated. For parameter values, the annotation is `__cdl(ValueInReference=false)` *while for other instances, the annotation is* `__cdl(InstanceInReference=false)`*. For both, if not specified, the default value is* `true`*.*

[A specification may look like

```
parameter Real anyOutOfScoMult(
  final unit = "1",
  final min = 0,
  final max = 1)=0.8
  "Outside of G36 recommended staging order chiller type SPLR multiplier"
  annotation(Evaluate=true, __cdl(ValueInReference=false));
```

]

Informative Note: This annotation is typically not provided for parameters that are in general not specified in ASHRAE Guideline 36, such as hysteresis deadband, default gains for a controller, or any reformulations of ASHRAE parameters that are needed for sequence generalization, for instance, a matrix variable used to indicate which chillers are used in each stage.

5.9 Extension Blocks

Extension blocks support functionalities that cannot, or is hard to, implement with a Composite Block. They allow implementation of blocks that contain statistical functions such as for regression, fault detection and diagnostics methods, or state machines for operation mode switches, as well as proprietary code. Extension blocks are also suited to document proposed new Elementary Blocks for later inclusion in ASHRAE Standard 231. In fact, Elementary Blocks are implemented using extension blocks, except that the `annotation __cdl(extensionBlock=true)` (see above) is not present.

In CDL, extension blocks must have the annotations:

```
annotation(__cdl(extensionBlock=true));
```

This annotation allows tools such as translators to recognize them as extension blocks. Extension blocks are equivalent to the class block in Modelica. Thus, extension blocks can contain any declarations that are allowed in a Modelica block.

Informative note: The fact that extension blocks allow any declaration that is allowed in a Modelica block implies that extension blocks can have any number of parameters, inputs, and outputs identical to Composite Blocks. It also implies that extension blocks can be used to

- *call code, for example, in C or from a compiled library,*

- *import a Functional Mockup Unit that may contain a process model or a fault detection and diagnostics method, and*
- *implement state machines.*

Translation of an extension block to CXF must reproduce the following:

- All parameters (except for protected parameters), inputs, and outputs.
- A Functional Mockup Unit for Model Exchange, version 2.0[1], with the file name being the full class name and the extension being `.fmu`.

Informative note: With OpenModelica 1.20.0, a Functional Mockup Unit for Model Exchange 2.0 of an extension block can be generated with the commands:

```
echo "loadFile(\"Buildings/package.mo\");" > translate.mos
echo "translateModelFMU(Buildings.Controls.OBC.CDL.Continuous.PID);" >>
translate.mos
omc translate.mos
```

This will generate the fmu Buildings.Controls.OBC.CDL.Continuous.PID.fmu.

5.10 Replaceable Blocks

CDL allows the use of the Modelica `replaceable`, `constrainedby` and `redeclare` keywords.

The `replaceable` keyword allows to replace a block by another block when translating a composite block.

To declare a block as `replaceable`, the syntax is

```
replaceable ClassName instanceName comment annotation;
```

where `ClassName` is the name of the class, `instanceName` is the name of the instance, and `comment` and `annotation` are optional comments or annotations.

Optionally, the `constrainedby` keyword can be added after `instanceName` to constrain what blocks can be used when redeclaring the replaceable block. The declaration is then

```
replaceable ClassName instanceName constrainedby NameOfConstrainingClass parameterBindings comment annotation;
```

where `NameOfConstrainingClass` is the name of the constraining class, and `parameterBindings` is optional and can be used to assign parameters, with or without the `final` keyword.

Informative note: For example, consider a composite block that has a PID controller. Suppose the developer of the composite block uses its custom PID controller called `MyPID`, and the developer wants to allow a user of the composite block to replace the PID controller with any custom PID controller, as long as it provides the inputs, outputs, and parameters of the elementary block of the PID controller `CDL.Reals.PID`.

Then, the composite block can be implemented as

[1] See https://fmi-standard.org

```
block SomeCompositeBlock "A composite block in a library"
  ...
  parameter Real k = 2 "Proportional gain";
  replaceable MyPID con constrainedby CDL.Reals.PID(k=k) "PID controller";
  ...
end SomeCompositeBlock;
```

Because of the `constrainedby` *clause, a user of the composite block can replace the controller* `MyPID` *with any other PID controller that also provides the inputs, outputs, and parameters that are present in* `CDL.Reals.PID`*. Moreover, the assignment* `k=k` *will also be applied when the controller is redeclared. Such a redeclaration in which a block* `MyPreferredPID` *is used for the instance* `con` *can be done using*

```
block SomeCompositeBlock "A composite block in a library"
  parameter Real k = 2 "Proportional gain";
  replaceable Buildings.Controls.OBC.CDL.Reals.PID conPID
     constrainedby Buildings.Controls.OBC.CDL.Reals.PID(k=k)
     "PID controller"
     annotation(Placement(transformation(
                              extent = {{-10, -10}, {10, 10}})));

  annotation(uses(Buildings(version = "12.0.0")));

end SomeCompositeBlock;
```

In a `redeclare` *statement, any parameters can be assigned, for example by writing* `redeclare MyPreferredPID conPID(Ti=60)`*, which sets the parameter* `Ti` *to* `60`*.*

The `constrainedby` *keyword can also be used to allow use of a block that has other parameters or inputs. A simple example is*

```
package ReplaceableExample
  block ReplaceableBlock
    replaceable Buildings.Controls.OBC.CDL.Reals.Sources.Constant con(k=1)
      constrainedby Buildings.Controls.OBC.CDL.Reals.Sources.CivilTime
     "Replaceable block, constrained by a block that imposes as a require-
ment that the redeclaration provides a block with output y (but no parameter
k is needed)";
  end ReplaceableBlock;

  block MyNewBlock "Composite block, with sou replaced by a Pulse with pe-
riod=0.1"
    ReplaceableBlock repBlo(
      redeclare Buildings.Controls.OBC.CDL.Reals.Sources.Pulse con(pe-
riod=0.1));
  end MyNewBlock;
  annotation (
    uses(Buildings(version = "12.0.0")));
end ReplaceableExample;
```

In the above code, the `constrainedby` *keyword specifies the block* `CivilTime`*.*

As `CivilTime` has only a `RealOutput` called `y`, but no parameters or inputs, the `Constant` block can be replaced by a `Pulse` block, although `Pulse` has no parameter `k`. Without the `constrainedby CDL.Reals.Sources.CivilTime` clause, `Pulse` could not have been used as it has no parameter `k`.

When translating CDL to CXF, the keywords `replaceable`, `constrainedby` and `redeclare` need to be evaluated and removed. E.g., they are not present in CXF.

5.11 Extension of a Composite Block

A composite block can have a single `extends` statement. The `extends` statement must reference another Composite Block, but it cannot extend an Elementary Block or an Extension Block. The `extends` statement can have any number of declarations that assign a parameter value or parameter attributes.

Informative note: There are three restrictions compared to the Modelica Language Specification:

- *Only a single `extends` statement is allowed. This is for simplicity because two `extends` statements could require having to reconcile two different hierarchy trees that ultimately extend from the same base block, but may assign different values to a parameter that is inherited from the common base block. Such a case would be for example*

```
package MultipleExtends
  block A0
    extends Buildings.Controls.OBC.CDL.Reals.Sources.Constant(k=0);
  end A0;

  block A1
    extends Buildings.Controls.OBC.CDL.Reals.Sources.Constant(k=1);
  end A1;

  block NotValid "Block that is not valid"
    extends A0;
    extends A1;
  end NotValid;

  annotation(
    uses(Buildings(version = "12.0.0")),
    Documentation(
      info = "<p>
Package with a block that is not valid CDL due to multiple extends state-
ments.
</p>"));

end MultipleExtends;
```

  Note that in Modelica, multiple `extends` are allowed, but the block `NotValid` is not valid and tools will issue an error message.
- *The break keyword for component deselection is not allowed.*
- *Modelica allows to assign a value to a variable declared as an `input`. This is not allowed in CDL. This restriction avoids that input connectors can no longer be*

graphically connected (as they then would have two bindings to a value, causing the block to be overdetermined).

Informative note: A simple illustrative example of an `extends` *statement would be to extends the block* `OBC.Utilities.PIDWithInputGains`*, restricts its output to be always between* `0` *and* `1`*, and adding an output connector that can be used to access the control error.*

This could be accomplished as

```
block MyPID
  extends Buildings.Controls.OBC.Utilities.PIDWithInputGains(
    final yMin = 0,
    final yMax = 1);

  Buildings.Controls.OBC.CDL.Interfaces.RealOutput error "Control error"
    annotation(Placement(
       transformation(origin = {240, -120},
        extent = {{-20, -20}, {20, 2
                        0}}),
       iconTransformation(origin = {120, -60},
        extent = {{-20, -20}, {20, 2
                        0}})));
equation
  connect(controlError.y, error)
    annotation(Line(
      points = {{-178, -6}, {-160, -6},{-160, -120}, {240, -120}},
      color = {0, 0, 127}));
  annotation(uses(Buildings(version = "12.0.0")),
    Documentation(info = "<p>
PID controller that extends the PID controller with input gains, and that
limits the output between 0 and 1, and adds an output connector that re-
ports
the control error.
</p>"));
end MyPID;
```

The `extends` statement can also have any number of `redeclare` statements (Section 5.10).
Informative note: For example, in the block below, the controller with name `conPID` *is replaced with the block* `OBC.CDL.Reals.PIDWithReset`*.*

```
model MyBlockWithRedeclare
  extends SomeCompositeBlock(
    redeclare Buildings.Controls.OBC.CDL.Reals.PIDWithReset conPID);
end MyBlockWithRedeclare;
```

5.12 Model of Computation

CDL uses the synchronous data flow principle and the single assignment rule, which are defined below.

1. All variables keep their actual values until these values are explicitly changed. Variable values can be accessed at any time instant.
2. Computation and communication at an event instant do not take time.

3. Every input connector shall be connected to exactly one output connector.

Informative note: The definition is adopted from and consistent with the Modelica Language Specification.

In addition, the dependency graph from inputs to outputs that directly depend on inputs shall be directed and acyclic; i.e., connections that form an algebraic loop are not allowed.

Informative note: To break an algebraic loop, one could place a delay block or an integrator in the loop because the outputs of a delay or integrator does not depend directly on the input.

5.13 Metadata

Informative Note: CDL provides the ability to embed semantic information which can be useful in documenting the control logic and integrating it with an existing building automation system. Examples of this include units, tags and descriptions. Formats for semantic information can be configured to comply with several open standards and schemas, including those from the proposed ASHRAE Standard 223[2], Project Haystack[3], and the BRICK consortium[4]. None of this information affects the computation of a control signal. Rather, it can be used for example to facilitate the implementation of cost estimation tools, or to detect incorrect connections between outputs and inputs.

5.13.1 Inferred Properties

Tools that translate CDL control logic to its CXF representation or tools that use CDL shall optionally infer the physical quantities from the unit and quantity attributes of the input and output connectors.

Informative note:
For example, a differential pressure input signal with name `u` can be declared as

```
Interfaces.RealInput u(
  quantity="PressureDifference",
  unit="Pa") "Differential pressure signal" annotation (...);
```

Examples of information that supports making such inferences:
Numerical value: a binary value (represented by a `Boolean` data type), an analog value (represented by a `Real` data type), and mode (represented by an `Integer` data type or an `Enumeration`).

- *Source: Hardware point or software point.*
- *Quantity: such as Temperature, Pressure, Humidity or Speed.*

[2] https://open223.info

[3] https://project-haystack.org

[4] https://brickschema.org

- *Unit: unit and display unit. (The display unit can be overwritten by a tool. This allows a control vendor to use the same sequences in North America displaying IP units, and in the rest of the world displaying SI units.)*

5.13.2 Semantic Information

CDL shall support the optional embedding of semantic information within the control logic. The semantic information shall be embedded and exported to a separate file using the specification mentioned in this section.

Informative Note: The information within a CDL control logic or the corresponding CXF representation includes all the details for a specific control sequence. This includes all the necessary blocks, the parameters of each of these blocks and how the blocks are connected through their input and output connectors for that sequence. But this control logic is only one part of the programming within the controller, and the controller is only one part of the control system. See Figure 8.1. To set up a complete control system, there is additional configuration required by the control contractor or systems integrator. Examples of the tasks involved in configuration are instantiating the control logic, connecting the right hardware input and output points to the corresponding connectors within the control logic, assigning BACnet objects (and correspondingly addresses), and connecting the logic in the control sequence to functions such as alarming, scheduling, and trending. Today, most of this configuration is done manually, although some vendors have tools to assist in the process. In the future, there may be standards which enable some or all of this process to be automated. The purpose of providing semantic information is to assist the person (or the tool) responsible for the configuration and integration to help reduce the complexity and potential errors in this process. The semantic information could also be used to export a semantic model or to represent the semantic requirements of a specific CDL control logic in a machine- (or human-) readable format.

Semantic information shall be included within the `annotation` keyword, using the `__cdl` annotation. `__cdl` shall be used when the semantic information is part of a control sequence. The following instances can optionally have annotations containing semantic information:

- input and output connectors
- parameters
- constants
- connections
- elementary blocks
- composite blocks
- extension blocks
- packages

All semantic information shall be included under the semantic section within the `__cdl` annotations using the syntax shown here:

```
annotation (__cdl(semantic(<semantic information>)));
```

where `<semantic information>` is a place holder for the semantic information.

The semantic annotation declared in the class definition of the CDL class can optionally contain the `metadataLanguageDefinition` or the `naturalLanguageDefinition` for each of the languages used. The `metadataLanguageDefinition` and `naturalLanguageDefinition` are used to provide additional information about the different metadata languages and natural languages that are used throughout the class. The language definitions shall contain information such as a short description of the language or the URL to the webpage of the language.

Informative Note: Even though `metadataLanguageDefinition` *and* `naturalLanguageDefinition` *are not mandatory, these definitions will improve the quality of the implementation because they provide additional information about the semantic standards used in the class.*

The optional `metadataLanguageDefinition` shall have the following syntax:

```
annotation (__cdl(semantic( metadataLanguageDefinition="<metadataLan-
guageName> <version> <format>" <"informative text">)));
```

where `<metadataLanguageName>` shall be replaced with the name of the metadata language, `<version>` is the mandatory entry for the version, `<format>` is the mandatory field for format of the language, such as text/turtle, and `<"informative text">` is a description of the language, such as the URL to the language. The version represents the version of the `<metadataLanguageName>` used in a particular class. The format represents the format that the semantic information is expressed in. The format shall be expressed using MIME types[5].

The optional `naturalLanguageDefinition` shall have the following syntax:

```
annotation (__cdl(semantic( naturalLanguageDefinition="<naturalLan-
guageName>" <"informative text">)));
```

where `<naturalLanguageName>` shall be replaced with the indicator of the natural language, represented using the ISO-639[6] language codes and `<"informative text">` is a description of the language. All `<naturalLanguageName>` metadata will be in the format `text/plain` MIME type.

Informative Note: Examples of the `<metadataLanguageName>` *include web ontology languages (OWL) such as Brick or ASHRAE 223p, and examples of* `<naturalLanguageName>` *include English (en) and Spanish (es). Below is an example of how to define multiple* `metadataLanguageDefinition` *and* `naturalLanguageDefinition` *in a class definition annotation.*

```
annotation (__cdl(semantic(
      metadataLanguageDefinition="Brick 1.3 text/turtle"
                  "https://brickschema.org/ontology/1.3",
      metadataLanguageDefinition="Project-Haystack 3.9.12
            application/ld+json"
                  "https://project-haystack.org/",
```

[5] https://www.iana.org/assignments/media-types/media-types.xhtml

[6] https://www.iso.org/iso-639-language-codes.html

```
        naturalLanguageDefinition="en" "Text in English language"
        )));
```

The semantic information shall be included as a `metadataLanguage/metadata` or a `naturalLanguage/metadata` pair within the semantic section in the `__cdl` annotation using the following syntax:

```
annotation (__cdl(semantic(metadataLanguage="<metadataLanguageName>
                           <version> <format>" "<metadata>")));

annotation (__cdl(semantic(naturalLanguage="<naturalLanguageName>"
                                            "<metadata>")));
```

where `<metadataLanguageName>` shall be replaced with the name of the metadata language, `<version>` is an entry for the version of the `metadataLanguage`, `<format>` is the format of the `metadataLanguage`, such as text/turtle, `<naturalLanguageName>` shall be replaced with the ISO-639 indicator of the natural language, and `<metadata>` is the metadata for that instance as specified in `<metadataLanguageName>` or `<naturalLanguageName>` language.

Informative Note: Depending on the `metadataLanguage` (*"*`<metadataLanguageName> <version> <format>`*"), the* `metadata` *can be represented in multiple formats. For example,* `text/turtle` *and* `application/ld+json` *are a couple of formats to represent the metadata of web ontology languages such as Brick and ASHRAE S223P. Project-Haystack* `metadata` *can also be represented in multiple formats such as* `text/zinc`*,* `text/turtle` *and* `application/ld+json`*.*

Semantic information in the class definition annotations shall be used to define class level information about the metadata languages.

Informative Note: Class level information about the metadata languages include, but are not restricted to, namespace definitions (namespaces in ontologies provide a means to unambiguously interpret identifiers and make the rest of the ontology presentation more readable) and prefixes (prefixes are shortcut abbreviations and help make the semantic information more readable). In the example below, for the `metadataLanguage` *"*`Brick 1.3 text/turtle`*", the class definition annotation has been used to define the namespace prefixes and for "*`Project-Haystack 3.9.12 application/ld+json`*", it has been used to define namespaces, prefixes, and contexts.*

```
annotation (__cdl(semantic(
      metadataLanguage="Brick 1.3 text/turtle"
            "@prefix Brick: <https://brickschema.org/schema/Brick#> .
             @prefix bldg: <urn:bldg/> . ",
      metadataLanguage="Project-Haystack 3.9.12 application/ld+json"
            "{\"@context\": { \"ph\": \"https://project-
                  haystack.org/def/ph/3.9.12#\",
              \"phScience\": \"https://project-
                  haystack.org/def/phScience/3.9.12#\",
              \"phIoT\": \"https://project-
                  haystack.org/def/phIoT/3.9.12#\",
```

```
                    \"rdf\": \"http://www.w3.org/1999/02/22-rdf-syntax-ns#\",
                    \"rdfs\": \"http://www.w3.org/2000/01/rdf-
                        schema#\"}}")));
```

If an instance declaration contains semantic information, it overrides the semantic information of its class definition. If an instance declaration does not contain semantic information, it inherits the semantic information of its class definition. Parameter (or constant) bindings can also have semantic information, and they override the semantic information of the parameter (or constant) declaration whose value is assigned.

Informative Note: Example of overriding semantic information is provided below.

```
CDL.Reals.MultiplyByParameter gain(k = 100000
      "My gain"
      annotation(__cdl(semantic(metadataLanguage="Brick 1.3
                  text/turtle" "<instance_name> a Brick:Gain_Parame-
            ter"))));
```

If there already exists a semantic model for a particular class or for an instance, it shall be referred to in the annotation using the syntax defined below:

```
annotation (__cdl(semantic(metadataLanguage="<metadataLanguageName>
      <version> <format>" "url=<path>")));

annotation (__cdl(semantic(naturalLanguage="<naturalLanguageName>"
      "url=<path>")));
```

where `<path>` shall be either a URL for a model that is on the network or a model that is present on the file system. If the `url=` is included in the metadata, the semantic model will be exported from `<path>`. If `url=` is not included in the metadata, `<path>` shall be the metadata.

If the metadata model is present on the file system as a separate file, the following syntax shall be followed:

```
annotation (__cdl(semantic(metadataLanguage="<metadataLanguageName>
            <version> <format>" "url=file:///<path/to/file>")));
annotation (__cdl(semantic(naturalLanguage="<naturalLanguageName>"
            "url=file:///<path/to/file>")));
```

Informative Note: Below are examples of how to refer to an existing "Brick 1.3 text/turtle" semantic model existing on the file system at "/home/user/soda_hall/soda_brick.ttl" and a "Project-Haystack 3.9.12 application/ld+json" semantic model on the network at the URL "https://project-haystack.org/example/download/alpha.jsonld":

```
annotation (__cdl( semantic( metadataLanguage="Brick 1.3 text/turtle"
      "url=file:///home/user/soda_hall/soda_brick.ttl")));

annotation (__cdl( semantic( metadataLanguage="Project-Haystack 3.9.12
                                          application/ld+json"
      "url=https://project-haystack.org/example/download
                                          /alpha.jsonld")));
```

`<instanceName>`: The text `<instanceName>` (including the < and > characters) within the metadata of an annotation containing semantic information shall be replaced with the fully qualified name of the instance that contains the semantic annotation. A fully qualified name to an instance refers to the complete hierarchical path that specifies the instance's location within an object structure. This qualified name shall include all parent instances leading up to the current instance, with each level of instantiation separated by an underscore ("_"). If an instance is nested within multiple levels of instance definitions, the text that replaces `<instanceName>` shall reflect the entire chain of instantiation. This avoids the user having to repeat the name of the instance and makes it less prone to errors and inconsistencies.

Informative Note: An example of CDL semantic information for an instance in a class with multiple `metadataLanguage/metadata` *pair is shown below. We can see that* `<instanceName>` *has been used in the* `metadata` *and Brick metadata will be inferred as* `bldg:THeaCoiSup_in a Brick:Hot_Water_Supply_Temperature_Sensor .` *and the Project Haystack identifier as* `{"@id": "THeaCoiSup_in"}` *assuming that the fully qualified path of* `THeaCoiSup_in` *is* `THeaCoiSup_in`.

Example:

```
Buildings.Controls.OBC.CDL.Interfaces.RealInput THeaCoiSup_in
      "Heating coil water supply temperature measurement"
      annotation (
            Placement(transformation(extent={{-140,-180},{-100,-140}})),
            __cdl(semantic(
            metadataLanguage="Brick 1.3 text/turtle"
                  "bldg:<instanceName> a
                        Brick:Hot_Water_Supply_Temperature_Sensor .",
            metadataLanguage=" Project-Haystack 3.9.12 application/ld+json"
                  "{
                     \"@id\": \"_:<instanceName> \",
                     \"ph:hasTag\": [
                        {\"@id\": \"phIoT:cur\"},
                        {\"@id\": \"phIoT:hot\"},
                        {\"@id\": \"phIoT:leaving\"},
                        {\"@id\": \"phIoT:point\"},
                        {\"@id\": \"phIoT:sensor\"},
                        {\"@id\": \"phScience:temp\"},
                        {\"@id\": \"phScience:water\"}
                     ],
                     \"rdfs:label\": \" Heating Hot Water Supply
                                       Temperature\" }",
            naturalLangauge="en" "<instanceName> is a temperature reading in-
      put that should be hardwired to heating coil temperature sensor")));
```

`<parameter>`: This syntax allows for a value of a parameter to be used within an annotation containing semantic information where the `parameter` shall refer to the name of a parameter instance within the class. The text `<parameter>` (including the < and > characters) shall be replaced by the value of the parameter. The class must have an instance of a parameter with the name specified by `<parameter>`, otherwise the specification is not valid.

Informative Note: In the example below, if the fully qualified name of `reaFloSup` *is* `reaFloSup`*, the* `<instanceName>` *will be replaced by* `reaFloSup`*. The location of the sensor, represented by the* `brick:hasLocation` *relationship, after replacing* `<instanceName>` *will be* `bldg:<zon>`*.* `<zon>` *refers to the value of the* `zon` *parameter within the instantiated* `reaFloSup`*, which is* `east`*. Hence, the completely evaluated semantic information becomes:*

```
bldg:reaFloSup a brick:Supply_Air_Flow_Sensor;
          brick:hasLocation bldg:east .
```

Example:

```
MyCompositeBlock.MyFlowSensor reaFloSup (zon="east") "Supply Air Flow Rate"
annotation ( __cdl(semantic(
    metadataLanguage="Brick 1.3 text/turtle"
        "bldg:<instanceName> a brick:Supply_Air_Flow_Sensor;
                  brick:hasLocation bldg:<zon> .")));
```

The semantic information of an instance shall be able to refer to the semantic information of other instances declared in the class. If the instance does not exist, the semantic model is invalid.

Informative Note: In the below example, the semantic information of heating coil `heaCoi` *is referring to the semantic information of the hot water supply temperature sensor* `THeaCoiSup_in`*.*

Example:

```
Modelica.Blocks.Interfaces.RealInput THeaCoiSup_in
      "Heating coil water supply temperature measurement"
      annotation (Placement(transformation(extent={{-140,-180},{-100,-
140}})),
                   __cdl(semantic(
                   metadataLanguage="Brick 1.3 text/turtle"
                  "bldg:<instanceName> a
                        Brick:Hot_Water_Supply_Temperature_Sensor ."
                  )));
Buildings.Fluid.HeatExchangers.DryCoilEffectivenessNTU heaCoi(
      show_T=true,
      dp1_nominal=3000,
      dp2_nominal=0 ) "Heating coil"
      annotation (Placement(transformation(extent={{118,-36},{98,-56}})),
             __Buildings(semantic(
                  metadataLanguage="Brick 1.3 text/turtle"
                        "bldg:<instanceName> a Brick:Heating_Coil ;
                              brick:hasPoint bldg:THeaCoiSup_in ."
                  )));
```

If a class inherits another class (CDL only allows for inheriting one class), all the semantic information in the classes is inherited. However, if the classes being inherited and the class inheriting it contains different `metadataLanguage` or `naturalLanguage` due to differences in any of `<metadataLanguageName>` or `<version>` or `<format>` or `<naturalLanguageName>` parts of the syntax, they shall be treated as different languages.

If an inherited `replaceable` instance has been replaced using the `redeclare` keyword, the semantic information of the instance that replaced the original instance shall be used, and the semantic information of the replaced class shall be ignored. If there is no semantic information in the redeclared instance annotation, any semantic information of the constraining clause (using the `constrainedby` Modelica keyword) of the original `replaceable` instance shall be used. Any semantic information in the original `replaceable` instance shall not be used if it has been redeclared irrespective of the presence or absence of semantic information in the constraining clause of the redeclared instance.

6 CONTROL EXCHANGE FORMAT (CXF)

6.1 Introduction

CXF is a representation of CDL in a format that is intended to be readily imported and exported into commercial building automation systems. For example, a commercial control provider might utilize CXF to import control logic from a design tool for deployment to their commercial building automation system for a particular project. Structurally, the content of a logic in CDL and CXF are identical, in that both utilize the same elementary blocks, composite blocks, and extension blocks as well as constants, parameters, input connectors, and output connectors. While CDL has language constructs that are used to build library of sequences, CXF was designed to only represent a specifically configured logic. The logic described in a CDL implementation is identical to the logic described in its CXF representation. But there are several key differences between CDL and CXF:

- CXF is defined utilizing the linked data format JSON-LD, while CDL utilizes the modeling language Modelica. JSON-LD is a syntax to serialize linked data in JSON (ECMA-404).
- There is a translation process required to convert a control logic from CDL to CXF.
- For ElementaryBlocks, their CXF representation does not include the implementation (equation section).
- Like many scientific modeling languages, Modelica requires tight casting of data types. *Informative Note:* For example, in Modelica, a data type needs to be declared as type `Real` or `Integer`. `Real` data are not allowed to be tested for equality since computations are prone to rounding errors.

  Informative Note:
  When importing a CXF representation of a CDL logic into a commercial control system that does not support `Real` *or* `Integer` *data types, the commercial entity's "CDL import" software tool must determine how to handle the* `Real` *and* `Integer` *InputConnectors, OutputConnectors, Parameters, and Constants. For example, the tool could change it to* `Analog`*. Similarly, while exporting a CXF representation of a control logic implemented in a commercial control system, the commercial entity's "CDL export" software tool must decide how to translate unsupported data types such as* `Analog` *into* `Real` *or* `Integer` *InputConnectors, OutputConnectors, Parameters, and Constants.*

- Control logic which utilizes arrays (both one- and multi -dimensional) in CDL shall have the option to be modified (or "flattened") in CXF (more details provided in a later section).

6.2 Classes and Properties

A valid CXF file contains Blocks (ElementaryBlocks, CompositeBlocks, ExtensionBlocks or a combination of these) and each instance of a Block uses the set of InputConnectors, OutputConnectors, Parameters, and Constants as defined within definition of the Block. To support the translation of a CDL control logic to its CXF representation, a Resource Description Framework graph representation of the standard has been provided in a CXF-Core.jsonld file using the MIME type `application/ld+json`. CXF-Core.jsonld. See the appendix for links to digital versions of this file. The key classes and properties present in CXF-Core.jsonld that can be used to create CXF classes are shown in Table 6-1 and Table 6-2 respectively.

Table 6-1Key classes within CXF-Core.jsonld

Class	Description
`Package`	`A Package is a specialized class used to group multiple Blocks.`
`Block`	`A Block is the abstract interface of a control logic.`
`ElementaryBlock`	`An ElementaryBlock defined by ASHRAE S231 (subClassOf Block)`
`CompositeBlock`	`A CompositeBlock is a collection of ElementaryBlocks, ExtensionBlocks or other CompositeBlocks (subClassOf Block) and the connections through their inputs and outputs.`
`ExtensionBlock`	`An ExtensionBlock supports functionalities that cannot, or are hard to, implement with a CompositeBlock (subClassOf Block).`
`InputConnector`	`An InputConnector provides an input to a Block.`

OutputConnector	An OutputConnector provides an output from a Block.
Parameter	A Parameter is a value that is time-invariant and cannot be changed based on an input signal.
Constant	A Constant is a value that is fixed at compilation time.
DataType	A data type description for InputConnectors, OutputConnectors, Parameters and Constants.
BooleanInput	An InputConnector of the Boolean data type.
BooleanOutput	An OutputConnector of the Boolean data type.
IntegerInput	An InputConnector of the Integer data type.
IntegerOutput	An OutputConnector of the Integer data type.
RealInput	An InputConnector of the Real data type.
RealOutput	An OutputConnector of the Real data type.
EnumerationType	An Integer enumeration starting with the value 1, each element is mapped to a unique String.
String	A data type to represent text.

Table 6-2: Key properties within CXF-Core.jsonld

Property	Domain	Range	Description
hasInput	Block	InputConnector	A property that relates an instance of an InputConnector with a Block.
hasOutput	Block	OutputConnector	A property that relates an instance of an OutputConnector with a Block.

hasParameter	Block	Parameter	A property that relates an instance of a Parameter with a Block.
hasConstant	Block	Constant	A property that relates an instance of a Constant with a Block.
hasInstance	Block	Block, InputConnector, OutputConnector, Parameter, Constant	A property that associates an instance of an InputConnector, OutputConnector, Parameter, Constant or a Block within a Block with the instance of the Block itself.
hasFmuPath	ExtensionBlock	String	A property that specifies the (local or on the network) path to a Functional Mockup Unit implementation of an ExtensionBlock.
isOfDataType	InputConnector, OutputConnector, Parameter, Constant	DataType	A property that specifies the data type for instances of InputConnectors, OutputConnectors, Parameters and Constants.
containsBlock	Block	Block	A property that specifies that an instance of a Block is composed in part with an instance of another Block.
connectedTo	OutputConnector, InputConnector	InputConnector, OutputConnector	A property that relates the output of one Block to the input of another Block (and vice-versa). Only InputConnectors and OutputConnectors that carry the same data type can be connected.
translationSoftware	Package, Block	String	A property that specifies the name of the software used to generate the CXF representation of the control logic.
translationSoftwareVersion	Package, Block	String	A property that specifies the version of the software used to generate CXF representation of the control logic.

6.3 Generating CXF from an instance of a CDL class

If the instantiation of a CDL block (within a Modelica or another CDL class) contains the annotation `__cdl(export=true)`, the CDL class of the instantiated block shall be translated to CXF. Specifying the `export` annotation is optional and if unspecified, `export=false` is assumed.

6.4 Source of a CXF translation

The CXF representation of a control logic shall optionally include the name and the version of the software that generated it using the properties `translationSoftware` and `translationSoftwareVersion` respectively.

6.5 Representing Instances in CXF

In the CXF representation of a CDL control logic, the instances of the CDL class shall contain the entire package path of the CDL class, the octothorpe character (`#`), followed by the name of the instance. An ("child") instance of an ("parent") instance shall be referenced in CXF by the parent instance's CXF representation, followed by a period character (`.`) and then the child instance's name. Additionally, the CXF representation of the parent instance shall contain a `hasInstance` property associating it to the child instance.

Informative Note: Example of a CDL instance representation in CXF

CDL:

```
    within ExamplePackage;
    block ExampleSeq
        CDL.Reals.MultiplyByParameter gain(k = 100000)
            "My gain";
    end ExampleSeq;
```

CXF reference to gain *instance:* ExamplePackage.ExampleSeq#gain
CXF reference to gain.k *instance:* ExamplePackage.ExampleSeq#gain.k
CXF property linking gain *and* gain.k*:*

```
        ExamplePackage.ExampleSeq#gain S231:hasInstance
                                    ExamplePackage.ExampleSeq#gain.k .
```

6.6 Handling Arrays and Expressions

Arrays and expressions in a CDL class shall be represented in CXF as specified below:

- Arrays (both one-dimensional (vectors) and multi-dimensional): In the CXF translation, array references shall either be preserved or flattened. If the array references are to be flattened, the indices appearing within square brackets (`[` and `]`) in CDL shall be appended with the underscore (_) character and each index shall be concatenated with the underscore character (_).

  Informative Note: For example, if the array references are preserved, `A[1]` *in CDL shall be referenced as* `A[1]` *in CXF. If they are flattened,* `A[1]` *shall be represented as* `A_1` *and* `B[1,2]` *shall be represented as* `B_1_2`.

  Array references in CDL shall be flattened in the row-major approach. Flattened array references shall be generated row-wise, starting from the left-most element of the first row to the right-most element of the first row, before advancing to the next row, until the right-most element of the last row.

If there already exists an instance in the CDL logic with the same name as a flattened array reference, then the translation process shall raise an error.

Informative Note: For example, if in a CDL class, there exists a parameter instance `A_1` *and a vector with 3 elements* `A[3]`*, upon flattening, references to the first element of the vector (*`A[1]`*) would become* `A_1`*. As this instance already exists, the CXF translator tool shall raise an error*

- Expressions: The CXF translation of a CDL control logic shall either preserve or evaluate all the expressions present in the CDL logic, such as those within assignment operations, conditional assignments, and arithmetic operations. By default, the expressions shall be preserved in the CXF representation. If the expressions must be evaluated and the expressions contain references to a parameter, the value of the parameter shall be used in evaluating the expression. If the expressions must be evaluated and expressions contain references to parameter(s) that do not have a value binding, then the translation process shall raise an error.

6.7 ExtensionBlocks

Instances of ExtensionBlocks within a CDL class shall contain the annotation `__cdl(extension=true)`. The location of the Functional Mockup Unit implementation of the ExtensionBlock shall be included in the CXF representation using the property `hasFmuPath`.

7 ELEMENTARY BLOCKS

7.1 Introduction

This standard includes a set of definitions for Elementary Blocks. The definition of an Elementary Block starts with the encoding (CDL or CXF), followed by the package name, which is used to structurally organize the blocks. See Figure 3. The full name is called the long class name, and the last part of the name (“Add”) is called the short class name.

Elementary Blocks are the elements that are at the root of this standard. Elementary Block definitions standardize their inputs, outputs, parameters, functionality. They also provide suggested graphical representation.

7.2 Specifying Elementary Blocks

Elementary Blocks have the same naming pattern as Composite Blocks and Extension Blocks, e.g., their long-qualified name starts with a list of package names, separated by a dot, and ends with the short name of the block. Elementary Blocks are stored in sub-packages of the CDL package. For example, the long name of the Add block for Reals and Integers is `CDL.Reals.Add` and `CDL.Integers.Add`. While here the package name is indicative of the data types of its inputs and outputs, the definition of the blocks declares the data type for each input, output, and parameter.

There are three elements required to properly apply the elementary blocks defined in this standard.

1. CDL: Indicates that the block follows the Controls Description Language.
2. Packages: For many blocks, the "package" name is used to describe the type of block. Examples of packages include Conversions, Integers, Psychrometrics, etc. Inputs, outputs, and parameters for blocks used in CDL will always be of the data types of Reals, Integers, or Booleans. The elements used in CXF can follow the allowed CDL options, as well as the use of Analogs in place of Reals or Integers. There are one or multiple package names that are appended to CDL or CXF. Package names are used to organize the blocks and to uniquely name them.
3. Elementary Function: The final part to defining a block is the elementary function. Elementary function names refer to the function of the block which is typically a mathematical or logical functions.

Example of a fully qualified name of an Elementary Block:

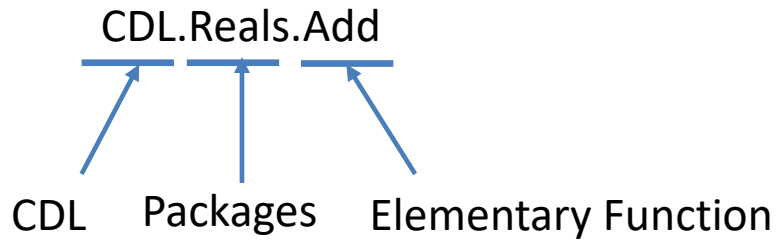


Figure 7-1: Naming for Elementary Blocks

Informational Notes:

- *Each description of Elementary Blocks includes a table that shows the required parameters, inputs and outputs.*
- *The term Boolean is used in this standard to refer to two state values such as true/false, which for some controllers may be understood as on / off or open / close. The term Binary is used to describe this in ASHRAE 135 (BACnet) and is also widely used in control systems. These terms are intended to represent two state actions; however, Boolean is the preferred term.*
- *Elementary blocks can also be represented in CXF. For more details refer to Section 6.*

7.3 Symbols

Each block is defined with a graphical representation (a symbol) which illustrates the function of that block. The use of these symbols is recommended but not required for compliance with the standard.

Symbol colors shall be coded as follows:

- If the block has any input being sampled, the block symbol is yellow, with the RGB value as {255,213,170}.
- If the block does not have any input being sampled and has Boolean connectors, the block symbol is gray, with the RGB value as {210,210,210}.
- If the block does not have any input being sampled and does not have Boolean connectors, the block is white, with the RGB values as {255,255,255}.

Table 7-1:Color codes for input and outputs of symbols

Connector		RGB code		
		Red	Green	Blue
`BooleanInput`	`LineColor`	255	0	255
	`FillColor`	255	0	255
`BooleanOutput`	`LineColor`	255	0	255
	`FillColor`	255	255	255
`IntegerInput`	`LineColor`	255	127	0
	`FillColor`	255	127	0
`IntegerOutput`	`LineColor`	255	127	0
	`FillColor`	255	255	255
`RealInput`	`LineColor`	0	0	127
	`FillColor`	0	0	127
`RealOutput`	`LineColor`	0	0	127
	`FillColor`	255	255	255

7.4 Elementary Blocks

Math Functions:

Name	Details	Data Types	Function
Abs	7.5.1	Real, Integer	Returns the absolute value of an analog value input
Add	7.5.2	Real, Integer	Adds two analog values
AddParame-ter	7.5.3	Real, Integer	Adds an analog value with a parameter value
Acos	7.5.4	Real	Returns the arccosine of an analog value: y = arccosine (u)
Asin	7.5.5	Real	Returns the arcsine of an analog value: y = arcsine (u)
Atan	7.5.6	Real	Returns the arctangent of an analog value: y = arctangent (u)

Atan2	7.5.7	Real	Returns the arctangent of two analog values: y = arctangent (u1 / u2)
Average	7.5.8	Real	Returns the average of analog values input. y=(u1+u2)/2
Cos	7.5.9	Real	Returns the cosine of an analog value: y = cosine (u)
Derivative	7.5.10	Real	Returns the approximate derivative of an input
Divide	7.5.11	Real	Divides two analog values as shown: y = u1 / u2
Exp	7.5.12	Real	Returns the base-e exponential of an analog value: y = exp (u)
Integrator-WithReset	7.5.13	Real	Integrates an input using a gain parameter. Has the ability to reset the integration using a boolean trigger.
LimitSlewRate	7.5.14	Real	Limits the increase or decrease rate of the input,
Line	7.5.15	Real	Accepts parameters that define two support points of a line. The line is then used to interpolate or extrapolate the input, optionally within an upper and lower bound.
Log	7.5.16	Real	Returns the natural log of an analog value: y = log(u) | u > 0
Log10	7.5.17	Real	Returns the base 10 log of an analog value: y = log10(u) | u > 0
MatrixGain	7.5.18	Real	Outputs the gain matrix with the input signal vector
MatrixMax	7.5.19	Real	Output vector for the maximum element of a matrix
MatrixMin	7.5.20	Real	Output vector for the minimum element of a matrix
Max	7.5.21	Real, Integer	Returns the maximum value of two inputs
Min	7.5.22	Real, Integer	Returns the minimum value of two inputs
Modulo	7.5.23	Real	Returns the remainder of first analog input divided by the second analog input
MovingAverage	7.5.24	Real	Function that outputs a moving average.

MultiMax	7.5.25	Real	Outputs the maximum element from an input vector.
MultiMin	7.5.26	Real	Outputs the minimum element from an input vector.
Multiply	7.5.27	Real, Integer	Returns the product of two inputs. u=x1*x2
MultiplyBy-Parameter	7.5.28	Real	Multiplies an analog value with a parameter value: y = k * u
MultiSum	7.5.29	Real, Integer	Multiplies each input with a parameter and outputs its sum, y = k[1]*u[1] + k[2]*u[2] + …… + k[n]*u[n]
	7.5.30	Real	PID Controller
PID-WithReset	7.5.31	Real	PID Controller with reset
Round	7.5.32	Real	Function that rounds a value to a specified number of digits
Sin	7.5.33	Real	Returns the sine of an analog value: y = sin (u)
Sort	7.5.34	Real	Function that sorts elements of a real input vector in ascending or descending order.
Sqrt	7.5.35	Real	Returns the square root of a value: y = sqrt(u) for u >= 0
Subtract	7.5.36	Real, Integer	Returns the difference of two values: y = u1 – u2
Tan	7.5.37	Real	Returns the tangent of an analog value: y = tan (u)

Logical Functions:

Name	Details	Data Types	Function
And	7.5.38	Boolean	Performs AND logic on two boolean inputs
Change	7.5.39	Integer, Boolean	Detects logical change of input in either direction
Edge	7.5.40	Boolean	Like ‘Change’ except only detects logical change of input from ‘false’ to ‘true’
FallingEdge	7.5.41	Boolean	Like ‘Edge’ except from ‘true’ to ‘false’
Latch	7.5.42	Boolean	When input goes ‘true’ output is ‘true’ unless reset is made ‘true’

MultiAnd	7.5.43	Boolean	Logical AND with more than two inputs
MultiOr	7.5.44	Boolean	Logical OR with more than two inputs
Nand	7.5.45	Boolean	Logical NAND on two boolean inputs: same as NOT [x AND y]
Nor	7.5.46	Boolean	logical NOR on two boolean inputs: same as NOT [x OR y])
Not	7.5.47	Boolean	Outputs the logical opposite of the input
Or	7.5.48	Boolean	Performs OR logic on two boolean inputs
Proof	7.5.49	Boolean	Verifies that feedback matches command
Switch	7.5.50	Real, Integer, Boolean	Outputs one of two boolean inputs based on boolean input
Toggle	7.5.51	Boolean	Changes the boolean output each time the boolean input changes from false to true; except when the boolean 'clr' input is true
VariableP-ulse	7.5.52	Boolean	Produces an output that cycles at a variable rate
Xor	7.5.53	Boolean	Performs XOR logic on two boolean inputs

Psychrometrics Functions:

Name	Details	Data Types	Function
DewPoint_TDry-BulPhi	7.5.54	Real	Calculates the dew point temperature from dry bulb temperature and relative humidity
Specifi-cEnthalpy_TDry-BulPhi	7.5.55	Real	Calculates specific enthalpy from dry bulb temperature and relative humidity
WetBulb_TDry-BulPhi	7.5.56	Real	Calculates the wet bulb temperature from dry bulb temperature and relative humidity

Comparisons:

Name	Details	Data Types	Function
Equal	7.5.57	Integer	Function that indicates when two input values are equal by making the output true
Greater	7.5.58	Real, Integer	When analog input 'u1' is greater than analog input 'u2' the output is true
GreaterEqual	7.5.59	Integer	When analog input 'u1' is greater than or equal to analog input 'u2' the output is true.
GreaterEqual-Threshold	7.5.60	Integer	When analog input 'u1' is greater than or equal to a fixed parameter value the output is true.
Greater-Threshold	7.5.61	Real, Integer	When analog input 'u1' is greater than a fixed parameter value the output is true.
Hysteresis	7.5.62	Real	Function that compares an analog input to analog parameters 'uHigh' and 'uLow' to determine the output.
Less	7.5.63	Real, Integer	When analog input 'u1' is less than analog input 'u2' the output is true.
LessEqual	7.5.64	Integer	When analog input 'u1' is less than or equal to analog input 'u2' the output is true
LessEqual-Threshold	7.5.65	Integer	When analog input 'u1' is less than or equal to a parameter value the output is true.
LessThreshold	7.5.66	Real, Integer	When analog input 'u1' is less than the parameter value the output is true.)
Limiter	7.5.67	Real	Compares an analog input against two parameters 'uMax' and 'uMin' and passes any value between these parameters or the limit value if the input is outside these bounds.
OnCounter	7.5.68	Boolean	Function that increments the analog output [starting from a pre-defined value] each time the boolean input changes to true; 'reset' restarts counter.
Pre	7.5.69	Boolean	Breaks loops by a small delay.
Ramp	7.5.70	Real	It has one boolean input `active` and a real input `u`. If the input `active` is `true`, the change of input `u` will be limited by the rate between `raisingSlewRate` and `fallingSlewRate`.

Stage	7.5.71	Integer	Outputs the total number of stages to be enabled. It has a numeric input (0-1) and each stage has a minimum hold time.

Conversions:

Name	Details	Data Types	Function
BooleanToInteger	7.5.72	Integer, Boolean	Converts a Boolean value to an Integer.
IntegerToReal	7.5.73	Integer, Boolean	Converts an Integer to a Real value.
RealToInteger	7.5.74	Real, Integer	Rounds a Real value to the nearest Integer.
BooleanToReal	7.5.75	Real, Boolean	Converts a Boolean value to a Real value

Timers:

Name	Details	Data Types	Function
Timer	7.5.76	Boolean	Boolean input starts timer, time output accumulates while input is true [0 when false], boolean output is true when threshold parameter [time limit] is exceeded
TimerAccumulating	7.5.77	Boolean	Boolean input starts timer, time output accumulates whenever input is true and retains value,boolean reset input resets accumulated time to 0, boolean output is true when threshold parameter [time limit] is exceeded)
TrueDelay	7.5.78	Boolean	When boolean input is true and the delay time parameter expires, the boolean output is true; boolean output is immediately false when logical input is false
TrueFalseHold	7.5.79	Boolean	Holds an output signal for at least a specified duration

Sources:

Name	Details	Data Types	Function
Sources.CalenderTime	7.5.80	Real	A block that outputs the current time, day, month and year

Sources.CivilTime	7.5.81	Real	A block that outputs the current system time.
Sources.Constant	7.5.82	Real, Integer, Boolean	A block that outputs a constant value
Sources.Pulse	7.5.83	Real, Integer, Boolean	A block that outputs a pulse signal
Sources.Ramp	7.5.84	Real	A block that outputs a ramp signal
Sources.SampleTrigger	7.5.85	Boolean	A block that outputs a signal that is only true at sample times (defined by parameter period) and is otherwise false.
Sources.Sin	7.5.86	Real	A block that outputs a sine signal
Sources.TimeTable	7.5.87	Real, Integer, Boolean	A block that outputs the result of a table look-up with respect to time

Discrete:

Name	Details	Data Types	Function
FirstOrderHold	7.5.88	Real	A block that outputs the extrapolation through the values of the last two sampled input signals.
Sampler	7.5.89	Real	A block that outputs the input signal, sampled at a sampling rate defined via a parameter
TriggeredMax	7.5.90	Real	A block that outputs the input signal whenever the trigger input signal is rising (i.e., trigger changes to true). The maximum, absolute value of the input signal at the sampling point is provided as the output signal.
TriggeredMovingMean	7.5.91	Real	A block that outputs the triggered moving mean value of an input signal. At the start of the simulation, and whenever the trigger signal is rising (i.e., the trigger changes to true), the block samples the input, computes the moving mean value over the past 'n' samples, and produces this value at its output.
TriggeredSampler	7.5.92	Real	A block samples the continuous input signal whenever the trigger input signal is rising (i.e., trigger changes from false to true) and provides the sampled input signal as output. Before

			the first sampling, the output signal is equal to the initial value defined via parameter 'y_start'.
UnitDelay	7.5.93	Real	Output 'y' is the input 'u' of the previous sample instant. Before the second sample instant, the output 'y' is identical to parameter 'y_start'.
ZeroOrderHold	7.5.94	Real	A block that outputs the sampled input signal at sample time instants. The output signal is held at the value of the last sample instant during the sample points. At initial time, the block feeds the input directly to the output.

Routing

Name	Details	Data Types	Function
ExtractSignal	7.5.95	Real, Integer, Boolean	Extract signals from the vector-valued input signal and transfer them to the vector-valued output signal. The extraction scheme is specified by a parameter.
Extractor	7.5.96	Real, Integer, Boolean	Extracts scalar signal from the vector-valued input signal vector dependent on input index
ScalarReplicator	7.5.97	Real, Integer, Boolean	Replicates an input signal to an array of 'nout' identical output signals.
VectorFilter	7.5.98	Real, Integer, Boolean	Filters a vector input of size 'nin' to a vector of size 'nout' given a Boolean mask 'msk'.
VectorReplicator	7.5.99	Real, Integer, Boolean	Replicates a vector input signal of size 'nin', to a matrix with 'nout' rows and 'nin' columns, where each row is duplicating the input vector.

Utilities:

Name	Details	Data Types	Function
Assert	7.5.100	N/A	Print a warning message when input becomes false

SunRiseSet	7.5.101	N/A	Outputs the next sunrise and sunset times in UTC.

Interfaces:

Name	Details	Data Types	Function
Input	7.5.103	Real, Integer, Boolean	Connector with one input signal of a specified data type
Output	7.5.104	Real, Integer, Boolean	Connector with one output signal of a specified data type

7.5 Elementary Block Descriptions

The following section includes a description of each Elementary Block including its functionality, symbol, parameters, inputs, and outputs. Compliance with this standard requires implementation of the Elementary Blocks which adhere to these descriptions.

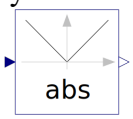
7.5.1 Abs

Outputs the absolute value of an input

Description: Block that outputs `y = abs(u)`, where u is an input.

7.5.1.1 CDL.Reals.Abs

Symbol



Parameters
N/A

Inputs

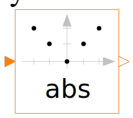
Data Types	Name	Description
Real	u	Input for absolute function

Outputs

Data Type	Name	Description
Real	y	Absolute value of the input

7.5.1.2 CDL.Integers.Abs

Symbol



Parameters
N/A

Inputs

Data Types	Name	Description
Integer	u	Input for absolute function

Outputs

Data Type	Name	Description
Integer	y	Absolute value of the input

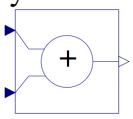
7.5.2 Add

Output the sum of the two inputs

Description: Block that outputs y as the sum of the two input signals u1 and u2, y = u1 + u2.

7.5.2.1 CDL.Reals.Add

Symbol



Parameters:
N/A

Inputs

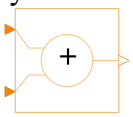
Data Types	Name	Description
Real	u1	Input to be added
Real	u2	Input to be added

Outputs

Data Type	Name	Description
Real	y	Sum of the two inputs

7.5.2.2 CDL.Integers.Add

Symbol



Parameters:
N/A

Inputs

Data Types	Name	Description
Integer	u1	Input to be added
Integer	u2	Input to be added

Outputs

Data Type	Name	Description
Integer	y	Sum of the two inputs

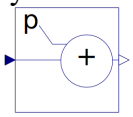
7.5.3 AddParameter

Outputs the sum of an input plus a parameter

Description: Block that outputs y = u + p, where p is parameter and u is an input.

7.5.3.1 CDL.Reals.AddParameter

Symbol



Parameters

Data Types	Name	Default	Description
Real	p	n/a	Parameter to be added to the input

Inputs

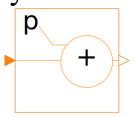
Data Types	Name	Description
Real	u	Input to be added to the parameter

Outputs

Data Type	Name	Description
Real	y	Sum of the parameter and the input

7.5.3.2 CDL.Integers.AddParameter

Symbol



Parameters

Data Types	Name	Default	Description
Integer	p	n/a	Parameter to be added to the input

Inputs

Data Types	Name	Description
Integer	u	Input to be added to the parameter

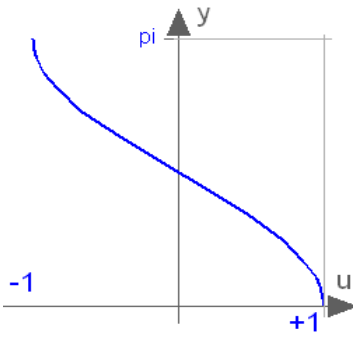
Outputs

Data Type	Name	Description
Integer	y	Sum of the parameter and the input

7.5.4 Acos

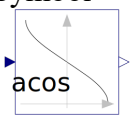
Outputs the arc cosine of the input

Description: Block that outputs y = acos(u), where u is an input.



7.5.4.1 CDL.Reals.Acos

Symbol



Parameters
N/A

Inputs

Data Types	Name	Description
Real	u	Input for arc cosine function

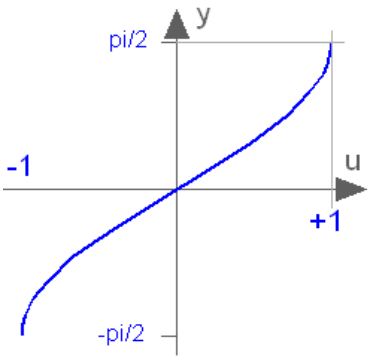
Outputs

Data Type	Name	Description
Real	y	Arc cosine of the input

7.5.5 Asin

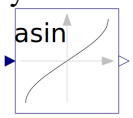
Outputs the arc sine of the input

Description: Block that outputs y = asin(u), where u is an input.



7.5.5.1 CDL.Reals.Asin

Symbol



Parameters:
N/A

Inputs

Data Types	Name	Description
Real	u	Input for the arc sine function

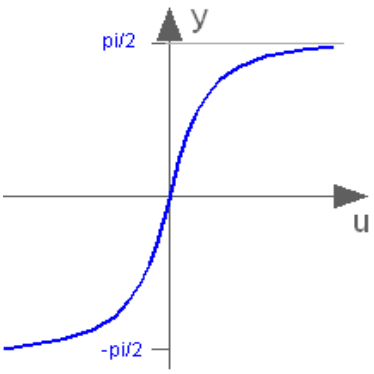
Outputs

Data Type	Name	Description
Real	y	Arc sine of the input

7.5.6 Atan

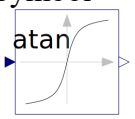
Outputs the arc tangent of the input

Description: Block that outputs y = atan(u), where u is an input



7.5.6.1 CDL.Reals.Atan

Symbol



Parameters
N/A

Inputs

Data Types	Name	Description
Real	u	Input for the arc tangent function

Outputs

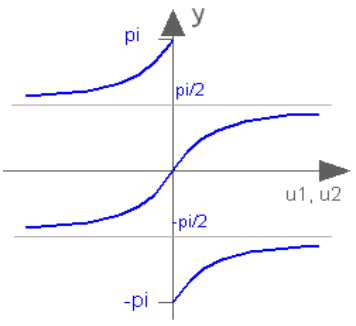
Data Type	Name	Description
Real	y	Arc tangent of the input

7.5.7 Atan2

Output atan(u1/u2) of the inputs u1 and u2

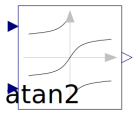
Description: Block that outputs the tangent-inverse y = atan2(u1, u2) of the input u1 divided by the input u2. u1 and u2 shall not be zero at the same time.

Informational Note:
Atan2 uses the sign of u1 and u2 in order to construct the solution in the range $-\pi \leq y \leq \pi$, whereas CDL.Reals.Atan gives a solution in the range $-\pi/2 \leq y \leq \pi/2$.



7.5.7.1 CDL.Reals.Atan2

Symbol



Parameters
N/A

Inputs

Data Types	Name	Description
Real	u1	Input u1 for the atan2(u1/u2) function
Real	u2	Input u2 for the atan2(u1/u2) function

Outputs

Data Type	Name	Description
Real	y	Output with atan2(u1/u2)

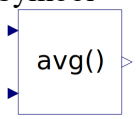
7.5.8 Average

Outputs the average of two inputs

Description: Block that outputs y = (u1 + u2) / 2, where u1 and u2 are inputs.

7.5.8.1 CDL.Reals.Average

Symbol



Parameters
N/A

Inputs

Data Types	Name	Description
Real	u1	Input for average function
Real	u2	Input for average function

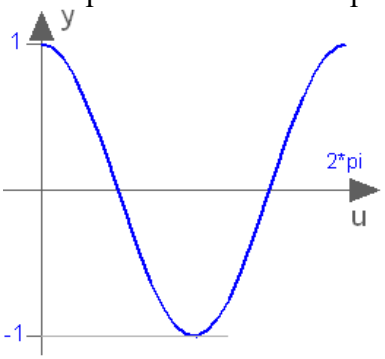
Outputs

Data Type	Name	Description
Real	y	Output with the average of the two inputs

7.5.9 Cos

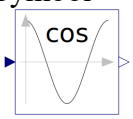
Outputs the cosine of an input

Description: Block that outputs y = cos(u), where u is an input.



7.5.9.1 CDL.Reals.Cos

Symbol



Parameters
N/A

Inputs

Data Types	Name	Description
Real	u	Input for the cosine function

Outputs

Data Type	Name	Description
Real	y	Cosine of the input

7.5.10 Derivative

Outputs the approximates the derivative of the input

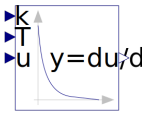
Description: This block defines the transfer function between the input `u` and the output `y` as *approximated derivative*:

$$y = k \frac{s}{T s + 1} u$$

If `k=0`, the block reduces to `y=0`.

7.5.10.1 CDL.Reals.Derivative

Symbol



Parameters

Data Types	Name	Default	Description
Real	y_start	0	Initial value of output (=state)

Inputs

Data Types	Name	Description
Real	k	Input for the gain
Real	T	Input for the time constant (T>0 required; T=0 is ideal derivative block) [s]
Real	u	Input to be differentiated

Outputs

Data Type	Name	Description
Real, Integer	y	Approximation of derivative du/dt

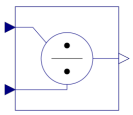
7.5.11 Divide

Output of first input divided by the second input

Description: Block that outputs y = u1/u2, where u1 and u2 are inputs.

7.5.11.1 CDL.Reals.Divide

Symbol



Parameters
N/A

Inputs

Data Types	Name	Description
Real	u1	Input for dividend
Real	u2	Input for divisor

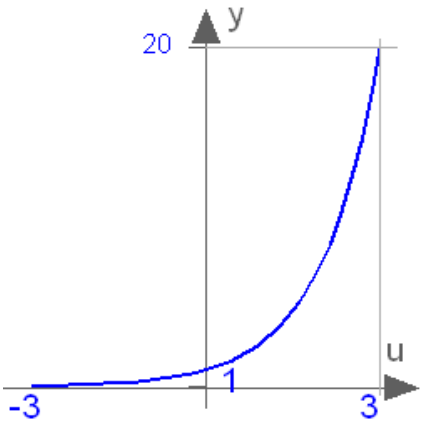
Outputs

Data Type	Name	Description
Real	y	Output with the quotient

7.5.12 Exp

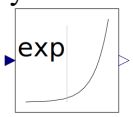
Outputs the exponential (base e) of the input

Description: Block that outputs y=exp(u), where u is an input and exp() is the base-e exponential function.



7.5.12.1 CDL.Reals.Exp

Symbol



Parameters
N/A

Inputs

Data Types	Name	Description
Real	u	Input for the base e exponential function

Outputs

Data Type	Name	Description
Real	y	Base e exponential value of the input

7.5.13 IntegratorWithReset

Outputs the integral of the input signal

Description: Block that outputs

$$y = y_{start} + \int_{t_{start}}^{t} u(s)\, ds$$

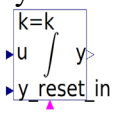
where y_{start} is a parameter, t_{start} is the time at which the integrator started, and u is the input.

The output of the integrator can be reset as follows:

Whenever the input signal trigger changes from false to true, the integrator is reset by setting y_{start} to the value of the input signal y_reset_in.

7.5.13.1 CDL.Reals.IntegratorWithReset

Symbol



Parameters

Data Types	Name	Default	Description
Real	k	1	Integrator gain
Note that for Simulation the following is required			
Real	y_start	0	Initial value of output (= state)

Inputs

Data Types	Name	Description
Boolean	trigger	Input that resets the integrator output when it becomes true
Real	u	Input to be integrated
Real	y_reset_in	Input signal for state to which integrator is re-set

Outputs

Data Type	Name	Description
Real	y	Value of the integrator

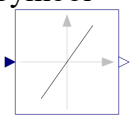
7.5.14 LimitSlewRate

Limit the rate of change of an input.

Description: Block that limits the rate of change of the input by a ramp. This block computes a threshold for the rate of change of the output y as thr = (u-y)/Td, where Td > 0 is parameter. The output y is computed as follows:
If thr < fallingSlewRate, then dy/dt = fallingSlewRate,
if thr > raisingSlewRate, then dy/dt = raisingSlewRate,
otherwise, dy/dt = thr.

7.5.14.1 CDL.Reals.LimitSlewRate

Symbol



Parameters

Data Types	Name	Default	Description
Boolean	enable	true	Set to false to disable rate limiter
Real	fallingSlewRate	-raisingSlewRate	Speed with which to decrease the output [1/s]
Real	raisingSlewRate	N/A	Speed with which to increase the output [1/s]
Real	Td	raisingSlewRate*10	Derivative Time Constant(s)

Inputs

Data Types	Name	Description
Real	u	Input signal to be rate of change limited

Outputs

Data Type	Name	Description
Real	y	Rate of change limited output signal

7.5.15 Line

Output the value of the input x along a line specified by two points

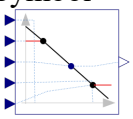
Description: Block that outputs y = a + b u, where u is an input and the coefficients a and b are determined so that the line intercepts the two input points specified by the two points x1 and f1, and x2 and f2.

The parameters limitBelow and limitAbove determine whether x1 and x2 are also used to limit the input u.

If the limits are used, then this block requires x1 < x2.

7.5.15.1 CDL.Reals.Line

Symbol



Parameters

Data Types	Name	Default	Description
Boolean	limitAbove	true	If true, limit input u to be no larger than x2
Boolean	limitBelow	true	If true, limit input u to be no smaller than x1

Inputs

Data Type	Name	Description
Real	f1	Input for support point f(x1)
Real	f2	Input for support point f(x2)
Real	u	Input for independent variable
Real	x1	Input for support point x1, with x1 < x2
Real	x2	Input for support point x2, with x2 > x1

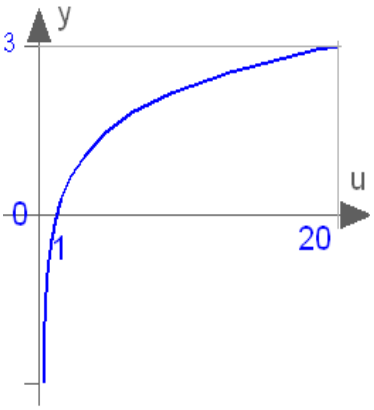
Outputs

Data Type	Name	Description
Real	y	Output with f(x) along the line specified by (x1, f1) and (x2, f2)

7.5.16 Log

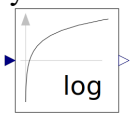
Output the natural (base e) logarithm of the input (input > 0 required)

Description: Block that outputs y = log(u), where u is an input and log() is the natural logarithm (base-e) function.



7.5.16.1 CDL.Reals.Log

Symbol



Parameters
N/A

Inputs

Data Type	Name	Description
Real	u	Input for base e logarithm

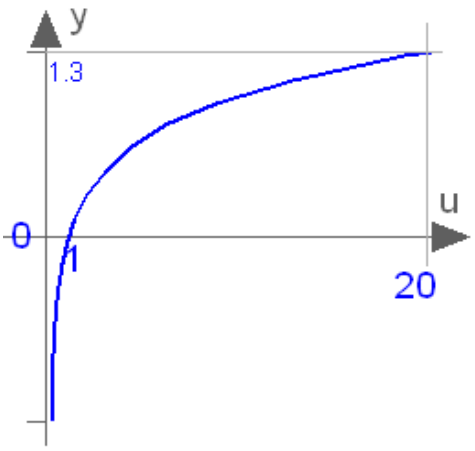
Outputs

Data Type	Name	Description
Real	y	Base e logarithm of the input

7.5.17 Log10

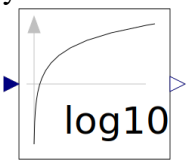
Output the base 10 logarithm of the input (input > 0 required)

Description: Block that outputs y = log10(u), where u is an input and log10() is the logarithm (base-10) function.



7.5.17.1 CDL.Reals.Log10

Symbol



Parameters
N/A

Inputs

Data Types	Name	Description
Real	u	Input for base 10 logarithm

Outputs

Data Type	Name	Description
Real	y	Base 10 logarithm of the input

7.5.18 MatrixGain

Outputs the product of a gain matrix with the input signal vector

Description: Block computes output vector *y* as the product of the gain matrix *K* with the input signal vector *u* as $y = K\,u$. For example,

parameter Real K[:, :] = [0.12, 2; 3, 1.5];

results in:

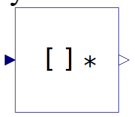
|y[1] | |0.12 2.00| |u[1]|

| | = | | * | |

|y[2] | |3.00 1.50| |u[2]|

7.5.18.1 CDL.Reals.MatrixGain

Symbol



Parameters

Data Types	Name	Default	Description
Real	K[:, :]	[1, 0; 0, 1]	Gain matrix which is multiplied with the input

Inputs

Data Types	Name	Description
Real	u[nin]	Input to be multiplied with the gain matrix

Outputs

Data Type	Name	Description
Real	y[nout]	Product of gain matrix times the input

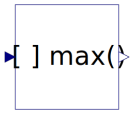
7.5.19 MatrixMax

Output vector of row- or column-wise maximum of the input matrix

Description: If rowMax = true, this block outputs the row-wise maximum of the input matrix u, otherwise it outputs the column-wise maximum of the input matrix u.

7.5.19.1 CDL.Reals.MatrixMax

Symbol



Parameters

Data Types	Name	Default	Description
Integer	nCol		Numbers of columns in input matrix
Integer	nRow		Number of rows in input matrix

Boolean	rowMax	true	If true, outputs row-wise maximum, otherwise column-wise

Inputs

Data Types	Name	Description
Real	u[nRow,nCol]	Input for the matrix max function

Outputs

Data Type	Name	Description
Real	y[if rowMax then size(u, 1) else size(u, 2)]	Output with vector of row- or column-wise maximum of the input matrix

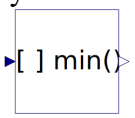
7.5.20 MatrixMin

Output vector of row- or column-wise minimum of the input matrix

Description: If rowMin = true, this block outputs the row-wise minimum of the input matrix u, otherwise it outputs the column-wise minimum of the input matrix u.

7.5.20.1 CDL.Reals.MatrixMin

Symbol



Parameters

Data Types	Name	Default	Description
Integer	nCol		Numbers of columns in input matrix
Integer	nRow		Number of rows in input matrix
Boolean	rowMin	true	If true, outputs row-wise minimum, other-wise column-wise

Inputs

Data Types	Name	Description
Real	u[nRow,nCol]	Input for the matrix min function

Outputs

Data Type	Name	Description
Real	y[if rowMin then size(u, 1) else size(u, 2)]	Output with vector of row- or colum-wise minimum of the input matrix

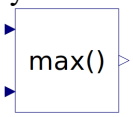
7.5.21 Max

Outputs the maximum of two inputs

Description: Block that outputs y = max(u1, u2), where u1 and u2 are inputs.

7.5.21.1 CDL.Reals.Max

Symbol



Parameters
N/A

Inputs

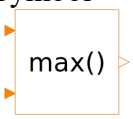
Data Types	Name	Description
Real	u1	Input to the max function
Real	u2	Input to the max function

Outputs

Data Type	Name	Description
Real	y	Maximum of the inputs

7.5.21.2 CDL.Integers.Max

Symbol



Parameters
N/A

Inputs

Data Types	Name	Description
Integer	u1	Input to the max function
Integer	u2	Input to the max function

Outputs

Data Type	Name	Description
Integer	y	Maximum of the inputs

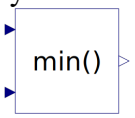
7.5.22 Min

Outputs the minimum of two inputs

Description: Block that outputs y = min(u1, u2), where u1 and u2 are inputs.

7.5.22.1 CDL.Reals.Min

Symbol



Parameters
N/A

Inputs

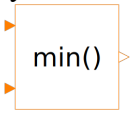
Data Types	Name	Description
Real	u1	Input to the min function
Real	u2	Input to the min function

Outputs

Data Type	Name	Description
Real	y	Minimum of the inputs

7.5.22.2 CDL.Integers.Min

Symbol



Parameters

N/A

Inputs

Data Types	Name	Description
Integer	u1	Input to the min function
Integer	u2	Input to the min function

Outputs

Data Type	Name	Description
Integer	y	Minimum of the inputs

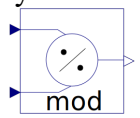
7.5.23 Modulo

Output the remainder of first input divided by second input

Description: Block that outputs y=mod(u1/u2), where u1and u2 are inputs. The input u2 must be non-zero.

7.5.23.1 CDL.Reals.Modulo

Symbol



Parameters
N/A

Inputs

Data Types	Name	Description
Real	u1	Dividend of the modulus function
Real	u2	Divisor of the modulus function

Outputs

Data Type	Name	Description
Real	y	Modulus u1 mod u2

7.5.24 MovingAverage

Outputs the moving average of an input

Description: Block which outputs the mean value of its input signal as:

$$y = \frac{1}{\delta} \int_{t-\delta}^{t} u(s)\, ds$$

where δ is a parameter that determines the time window over which the input is averaged. For $t < \delta$ seconds, it outputs:

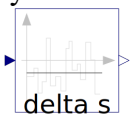
$$y = \frac{1}{t - t_0 + 10^{-10}} \int_{t_0}^{t} u(s)\, ds$$

where t_0 is the initial time.

This block can be used to output the moving average of a noisy measurement signal.

7.5.24.1 CDL.Reals.MovingAverage

Symbol



Parameters

Data Types	Name	Default	Description
Real	delta		Time horizon over which the input is averaged

Inputs

Data Types	Name	Description
Real	u	Input to be averaged

Outputs

Data Type	Name	Description
Real	y	Moving average of the input

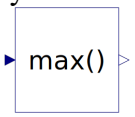
7.5.25 MultiMax

Outputs the maximum element of an input vector

Description: Block that evaluates an input vector u[:] and outputs the maximum value

7.5.25.1 CDL.Reals.MultiMax

Symbol



Parameters

Data Types	Name	Default	Description
Integer	nin	0	Number of input connections

Inputs

Data Types	Name	Description
Real	u[nin]	Input to max function

Outputs

Data Type	Name	Description
Real	y	Largest element of the input vector

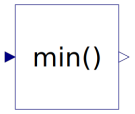
7.5.26 MultiMin

Outputs the minimum element of an input vector

Description: Block that evaluates an input vector u[:] and outputs the minimum value.

7.5.26.1 CDL.Reals.MultiMin

Symbol



Parameters

Data Types	Name	Default	Description
Integer	nin	0	Number of input connections

Inputs

Data Types	Name	Description
Real	u[nin]	Input to the min function

Outputs

Data Type	Name	Description
Real	y	Smallest element of the input vector

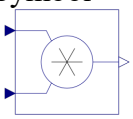
7.5.27 Multiply

Outputs the product of two inputs

Description: Block that outputs y = u1 * u2, where u1 and u2 are inputs.

7.5.27.1 CDL.Reals.Multiply

Symbol



Parameters
N/A

Inputs

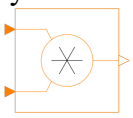
Data Types	Name	Description
Real	u1	Input to be multiplied
Real	u2	Input to be multiplied

Outputs

Data Type	Name	Description
Real	y	Product of the inputs

7.5.27.2 CDL.Integers.Multiply

Symbol



Parameters
N/A

Inputs

Data Types	Name	Description
Integer	u1	Input to be multiplied

Integer	u2	Input to be multiplied

Outputs

Data Type	Name	Description
Integer	y	Product of the inputs

7.5.28 MultiplyByParameter

Elementary Block Names

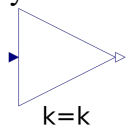
CDL	**CXF**
CDL.Reals.MultiplyByParameter	CXF.Reals.MultiplyByParameter
	CXF.Analogs.MultiplyByParameter

Output the product of a gain value with the input signal

Description: Block that outputs y = k * u, where k is a parameter and u is an input.

7.5.28.1 CDL.Reals.MultiplyByParameter

Symbol



Parameters

Data Types	Name	De-fault	Description
Real	k	N/A	Gain value to be multiplied with the input

Inputs

Data Types	Name	Description
Real	u	Input to be multiplied with gain

Outputs

Data Type	Name	Description
Real	y	Product of the parameter times the input

7.5.29 MultiSum

Sum of inputs, y = k[1]*u[1] + k[2]*u[2] + ... + k[n]*u[n]

Description: Block that outputs:

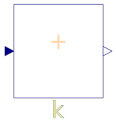
$$y = \sum_{i=1}^{n} (k_i \cdot u_i)$$

where *k* is a parameter with *n* elements and *u* is an input of the same length.

If no connection to the input connector *u* is present, the output is *y=0*.

7.5.29.1 CDL.Reals.Multisum

Symbol



Parameters

Data Types	Name	Default	Description
Real	k[nin]	Fill(1,nin)	Input gains
Integer	nin	0	Number of input connections

Inputs

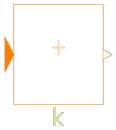
Data Types	Name	Description
Real	u[nin]	Input to multiplied by gain and then added

Outputs

Data Type	Name	Description
Real	y	Sum of inputs times gains

7.5.29.2 CDL.Integers.Multisum

Symbol



Parameters

Data Types	Name	Default	Description
Integer	k[nin]	Fill(1,nin)	Input gains
Integer	nin	0	Number of input connections

Inputs

Data Types	Name	Description
Integer	u[nin]	Input to multiplied by gain and then added

Outputs

Data Type	Name	Description
Integer	y	Sum of inputs times gains

7.5.30 PID

Block that utilizes proportional, proportional integral, or proportional integral derivative control

Description: PID in the standard form $y_u = k/r \ (e(t) + 1/T_i \int e(\tau)\, d\tau + T_d\, d/dt\, e(t))$, where y_u is the control signal before output limitation, $e(t) = u_s(t) - u_m(t)$ is the control error, with u_s being the set point and u_m being the measured quantity, k is the gain, T_i is the time constant of the integral term, T_d is the time constant of the derivative term, and r is a scaling factor, with default $r=1$.

Informative Notes:

- *The scaling factor should be set to the typical order of magnitude of the range of the error e. For example, you may set r=100 to r=1000 if the control input is a pressure of a heating water circulation pump in units of Pascal or leave r=1 if the control input is a room temperature.*

- *The units of k are the inverse of the units of the control error, while the units of T_i and T_d are seconds.*
- *The actual control output is $y = \min(y_{max}, \max(y_{min}, y))$, where y_{min} and y_{max} are limits for the control signal.*

P, PI, PD, or PID action:

Through the parameter controllerType, the block can be configured as P, PI, PD, or PID. The default configuration is PI.

Reverse or direct action

Through the parameter reverseActing , the block shall be configured to be reverse or direct acting. The above standard form is reverse acting, which is the default configuration. For reverse acting, for a constant set point, an increase in measurement signal u_m decreases the control output signal y. For a heating coil with a two-way valve, leave reverseActing = true , but for a cooling coil with a two-way valve, set reverseActing = false.
If reverseActing = false, then the error *e* above is multiplied by *-1*.

Anti-windup compensation
Anti-windup compensation is as follows: Instead of the above basic control law, the implementation is $y_u = k \ (e(t)/r + 1/T_i \int (-\Delta y + e(\tau)/r)\, d\tau + T_d/r\, d/dt\, e(t))$,
where the anti-windup compensation Δy is $\Delta y = (y_u - y)/(k N_i)$,
where $N_i > 0$ is the time constant for the anti-windup compensation. To accelerate the anti-windup, decrease N_i.

Informational Notes:

The anti-windup term $(-\Delta y + e(\tau)/r)$ shows that the range of the typical control error r should be set to a reasonable value so that $e(\tau)/r = (u_s(\tau) - u_m(\tau))/r$ has order of magnitude one, and hence the anti-windup compensation should work well.

Reset of the block output

This block implements an integrator anti-windup. Therefore, for most applications, the block output does not need to be reset. However, if the block is used in conjunction with equipment that is being switched on, better control performance may be achieved by resetting the PID block output when the equipment is switched on. This is the case in situations where the equipment control input should continuously increase as a variable speed drive of a motor that should continuously increase the speed. In this case, the function PIDWithReset that can reset the output should be used.

Approximation of the derivative term

The derivative of the control error $d/dt\, e(t)$ is approximated using $d/dt\, x(t) = (e(t)-x(t))\, N_d/T_d$, and $d/dt\, e(t) \approx N_d\, (e(t)-x(t))$, where x(t) is an internal state.

Guidance for tuning gains

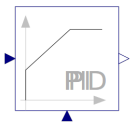
The parameters of the block can be manually adjusted by performing closed loop tests and using the following strategy:

1. *Set very large limits, e.g., set $y_{max} = 1000$.*
2. *Select a **P**-block and manually enlarge the parameter k (the total gain) until the closed-loop response cannot be improved any more.*
3. *Select a **PI**-block and manually adjust the parameters k and Ti (the time constant of the integrator). The first value of Ti can be selected such that it is in the order of the time constant of the oscillations occurring with the P-controller. If, e.g., oscillations in the order of 100 seconds occur in the previous step, start with Ti=1/100 seconds.*

4. *If you want to make the reaction of the control loop faster (but probably less robust against disturbances and measurement noise) select a **PID**-block and manually adjust parameters k, Ti, Td (time constant of derivative block).*
5. *Set the limits yMax and yMin according to your specification.*
6. *Perform simulations such that the output of the PID block goes in its limits. Tune Ni (N_i T_i is the time constant of the anti-windup compensation) such that the input to the limiter block (= lim.u) goes quickly enough back to its limits. If Ni is decreased, this happens faster. If Ni is very large, the anti-windup compensation is not effective.*

7.5.30.1 CDL.Reals.PID

Symbol



Parameters

Data Types	Name	Default	Description
Configuration			
CDL.Types.SimpleController	controllerType	PI	Type of controller (P, PI, PD, PID)
Real	r	1	Typical range of control error, used for scaling the control error
Boolean	reverseActing	true	Set to true for reverse acting, or false for direct acting control action
Control gains			
Real	k	1	Gain of controller
Real	Ti	0.5	Time constant of integrator block [s]
Real	Td	0.1	Time constant of derivative block [s]
Limits			
Real	yMax	1	Upper limit of output
Real	yMin	0	Lower limit of output
Advanced			
Integrator anti-windup			
Real	Ni	0.9	Ni*Ti is time constant of anti-windup compensation
Derivative block			
Real	Nd	10	The higher Nd, the more ideal the derivative block

Initialization			
Real	xi_start	0	Initial value of integrator state
Real	yd_start	0	Initial value of derivative output

Informational Note: It is recommended that at minimum the parameters for gains and limits be configurable while the block is running.

Inputs

Data Types	Name	Description
Real	u_m	Measurement input signal
Real	u_s	Setpoint input signal

Outputs

Data Type	Name	Description
Real	y	Connector of output signals (to actuator)

7.5.31 PIDWithReset

Block that utilizes proportional, proportional integral, or proportional integral derivative control with reset

Description: PID block in the standard form $y_u = k/r \ (e(t) + 1/T_i \int e(\tau)\, d\tau + T_d\, d/dt\, e(t))$, where y_u is the control signal before output limitation, $e(t) = u_s(t) - u_m(t)$ is the control error, with u_s being the set point and u_m being the measured quantity, k is the gain, T_i is the time constant of the integral term, T_d is the time constant of the derivative term, and r is a scaling factor, with default $r=1$.

Informational Notes:

- *The scaling factor should be set to the typical order of magnitude of the range of the error e. For example, you may set r=100 to r=1000 if the control input is a pressure of a heating water circulation pump in units of Pascal or leave r=1 if the control input is a room temperature.*

- *Note that the units of k are the inverse of the units of the control error, while the units of T_i and T_d are seconds.*

- *The actual control output is $y = \min(y_{max}, \max(y_{min}, y))$, where y_{min} and y_{max} are limits for the control signal.*

P, PI, PD, or PID action:

Through the parameter controllerType , the block can be configured as P, PI, PD, or PID block. The default configuration is PI.

Reverse or direct action
Through the parameter reverseActing, the block can be configured to be reverse or direct acting. The above standard form is reverse acting, which is the default configuration. For a reverse acting block, for a constant set point, an increase in measurement signal u_m decreases the control output signal y (Montgomery and McDowall, 2008). Thus,
for a heating coil with a two-way valve, leave reverseActing = true , but
for a cooling coil with a two-way valve, set reverseActing = false.
If reverseActing = false, then the error *e* above is multiplied by *-1*.

Anti-windup compensation
The block anti-windup compensation is as follows: Instead of the above basic control law, the implementation is $y_u = k \ (e(t) / r + 1 / T_i \ \int (-\Delta y + e(\tau) / r) \, d\tau + T_d / r \, d/dt \, e(t))$,
where the anti-windup compensation Δy is $\Delta y = (y_u - y) / (k \, N_i)$,
where $N_i > 0$ is the time constant for the anti-windup compensation. To accelerate the anti-windup, decrease N_i.

Informational Notes:

The anti-windup term $(-\Delta y + e(\tau) / r)$ shows that the range of the typical control error r should be set to a reasonable value so that $e(\tau) / r = (u_s(\tau) - u_m(\tau)) / r$ has order of magnitude one, and hence the anti-windup compensation should work well.

Reset of the block output
This block implements an integrator anti-windup. Therefore, for most applications, the block output does not need to be reset.

Reset of the block output
Whenever the value of boolean input signal trigger changes from false to true, the block output is reset by setting y to the value of the parameter y_reset.

Approximation of the derivative term
The derivative of the control error $d / dt \, e(t)$ is approximated using $d/dt \, x(t) = (e(t)-x(t)) \, N_d / T_d$, and $d/dt \, e(t) \approx N_d \, (e(t)-x(t))$, where x(t) is an internal state.

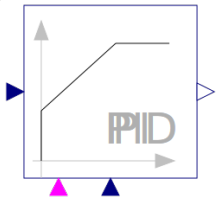
Guidance for tuning the control gains:
The parameters of the block can be manually adjusted by performing closed loop tests (= block + plant connected) and using the following strategy:

1. *Set very large limits, e.g., set $y_{max} = 1000$.*
2. *Select a **P**-block and manually enlarge the parameter k (the total gain of the block) until the closed-loop response cannot be improved any more.*

3. *Select a **PI**-block and manually adjust the parameters k and Ti (the time constant of the integrator). The first value of Ti can be selected such that it is in the order of the time constant of the oscillations occurring with the P-block. If, e.g., oscillations in the order of 100 seconds occur in the previous step, start with Ti=1/100 seconds.*
4. *If you want to make the reaction of the control loop faster (but probably less robust against disturbances and measurement noise), select a **PID**-block and manually adjust parameters k, Ti, Td (time constant of derivative block).*
5. *Set the limits yMax and yMin according to your specification.*
6. *Perform simulations such that the output of the PID block goes in its limits. Tune Ni (N_i T_i is the time constant of the anti-windup compensation) such that the input to the limiter block (= lim.u) goes quickly enough back to its limits. If Ni is decreased, this happens faster. If Ni is very large, the anti-windup compensation is not effective.*

7.5.31.1 CDL.Reals.PIDWithReset

Symbol



Parameters

Data Types	Name	Default	Description
Configuration			
CDL.Types.SimpleController	controllerType	PI	Type of controller (P, PI, PD, PID)
Real	r	1	Typical range of control error, used for scaling the control error
Boolean	reverseActing	true	Set to true for reverse acting, or false for direct acting control action
Control gains			
Real	k	1	Gain of controller
Real	Ti	0.5	Time constant of integrator block [s]
Real	Td	0.1	Time constant of derivative block [s]
Limits			
Real	yMax	1	Upper limit of output
Real	yMin	0	Lower limit of output
Integrator reset			

Real	y_reset	xi_start	Value to which the controller output is reset if the boolean trigger has a rising edge
Advanced			
Integrator anti-windup			
Real	Ni		Ni*Ti is time constant of anti-windup compensation
Derivative block			
Real	Nd	10	The higher Nd, the more ideal the derivative block
Initialization			
Real	xi_start	0	Initial value of integrator state
Real	yd_start	0	Initial value of derivative output

Informational Note: It is recommended that at minimum the parameters for gains and limits be configurable while the block is running.

Inputs

Data Types	Name	Description
Boolean	trigger	Resets the controller output when trigger becomes true
Real	u_m	Connector of the measurement input signal
Real	u_s	Connector for the setpoint input signal

Outputs

Data Type	Name	Description
Real	y	Connector of output signals (to actuator)

7.5.32 Round

Rounds an input to a specific number of digits.

Description: Block that outputs the input after rounding it to n digits.

For example,

- set n = 0 to round to the nearest integer,
- set n = 1 to round to the next decimal point, and
- set n = -1 to round to the next multiple of ten.

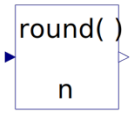
Hence, the block outputs

y = floor(u*(10^n) + .5)/10^n) for u > 0

y = ceil(u*(10^n) - .5)/10^n) for u < 0

7.5.32.1 CDL.Reals.Round

Symbol



Parameters

Data Types	Name	Default	Description
Real	n		Number of digits being round to

Inputs

Data Types	Name	Description
Real	u	Input to be rounded

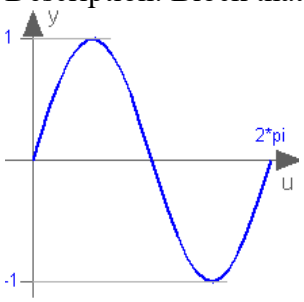
Outputs

Data Type	Name	Description
Real	y	Output with rounded input

7.5.33 Sin

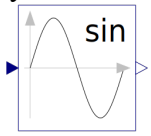
Outputs the sine of an input

Description: Block that outputs `y = sin(u)`, where u is an input.



7.5.33.1 CDL.Reals.Sin

Symbol



Parameters
N/A

Inputs

Data Types	Name	Description
Real	u	Input for the Sine function

Outputs

Data Type	Name	Description
Real	y	Sine of the input

7.5.34 Sort

Sort elements of input vector in ascending or descending order

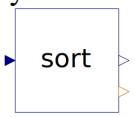
Description: Block that sorts the elements of the input signal *u*. If the parameter ascending = true, then the output signal *y* satisfies $y_i <= y_{i+1}$ for all $i \in \{1, ..., n-1\}$. Otherwise, it satisfies $y_i >= y_{i+1}$ for all $i \in \{1, ..., n-1\}$. The output signal *yIdx* contains the indices of the sorted elements with respect to the input vector *u*.

Informational Notes:
Note that this block is used for input signals u that are time sampled. In simulation, numerical noise from a nonlinear solver or from an implicit time integration algorithm may cause the simulation to stall. Numerical noise can be present if an input depends on a state variable or a quantity that requires an iterative solution, such as a temperature or a mass flow rate of an HVAC system. In real controllers, measurement noise may cause the output to change frequently.

This block may for example be used in a variable air volume flow controller to access the position of the dampers that are most open.

7.5.34.1 CDL.Reals.Sort

Symbol



Parameters

Data Types	Name	Default	Description
Boolean	ascending	true	Set to true if ascending order, otherwise order is descending
Integer	nin	n/a	Number of input connectors

Inputs

Data Types	Name	Description
Real	u[nin]	Input(s) to be sorted

Outputs

Data Type	Name	Description
Real	y[nin]	Outpur of sorted inouts
Integer	yIdx[nin]	Indices of the sorted vector with respect to the original vector

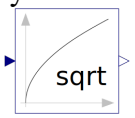
7.5.35 Sqrt

Outputs the square root of an input (input > = 0 required)

Description: Block which outputs the square root of the input `y = sqrt(u)`, where `u` is an input. The input u shall be non-negative.

7.5.35.1 CDL.Reals.Sqrt

Symbol



Parameters
N/A

Inputs

Data Types	Name	Description
Real	u	Input to square root function

Outputs

Data Type	Name	Description
Real	y	Output with square root of the input

7.5.36 Subtract

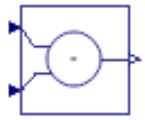
Outputs the difference of two inputs

Description: Block that outputs `y` as the difference of the two input signals `u1` and `u2`,

```
y = u1 - u2
```

7.5.36.1 CDL.Reals.Subtract

Symbol



Parameters
N/A

Inputs

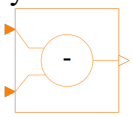
Data Types	Name	Description
Real	u1	Input with minuend
Real	u2	Input with subtrahend

Outputs

Data Type	Name	Description
Real	y	Output with difference

7.5.36.2 CDL.Integers.Subtract

Symbol



Parameters
N/A

Inputs

Data Types	Name	Description
Integer	u1	Input with minuend
Integer	u2	Input with subtrahend

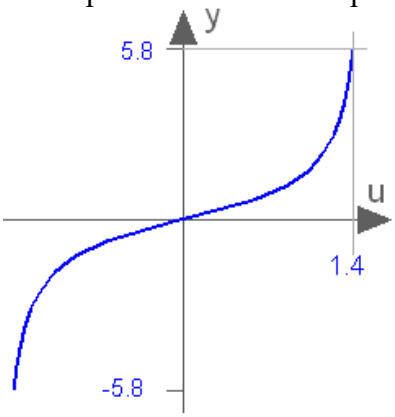
Outputs

Data Type	Name	Description
Integer	y	Output with difference

7.5.37 Tan

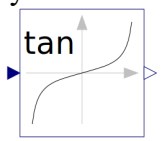
Outputs the tangent of an input

Description: Block that outputs y = tan(u), where u is an input.



7.5.37.1 CDL.Reals.Tan

Symbol



Parameters
N/A

Inputs

Data Types	Name	Description
Real	u	Input for the tangent function

Outputs

Data Type	Name	Description
Real	y	Tangent of the input

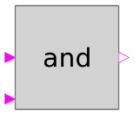
7.5.38 And

Logical 'and': y = u1 and u2

Description: Block that outputs true if all inputs are true, else it outputs false

7.5.38.1 CDL.Logical.And

Symbol



Parameters
N/A

Inputs

Data Types	Name	Description
Boolean	u1	Input signal for logical 'and'
Boolean	u2	Input signal for logical 'and'

Outputs

Data Type	Name	Description
Boolean	y	Outputs true if u1 and u2 are both true

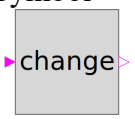
7.5.39 Change

Output whether the input changes values, increases or decreases

Description: For package of type Logical, if the input has either a rising edge from false to true or a falling edge from true to false, the block outputs true. Otherwise, the output is false. For package of type Integer, if the input increases or decreases value, the block outputs true. Otherwise, the output is false. Different from the package of type Logical, the block in package of type Integer has two additional outputs that indicate the direction of change.

7.5.39.1 CDL.Logical.Change

Symbol



Parameters

Data Types	Name	Default	Description
Boolean	pre_u_start	(false)	For simulation, this parameter is used as the start value of pre(u).

Inputs

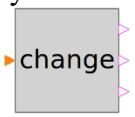
Data Types	Name	Description
Boolean	u	Input to be monitored for a change

Outputs

Data Type	Name	Description
Boolean	y	Output which is true when the input changes

7.5.39.2 CDL.Integer.Change

Symbol



Parameters

Data Types	Name	Default	Description
Integer	pre_u_start	(0)	For simulation, this parameter is used as the start value of pre(u).

Inputs

Data Types	Name	Description
Integer	u	Input to be monitored for a change in value

Outputs

Data Type	Name	Description
Boolean	down	Output that is true when the input decreased its value
Boolean	up	Output that is true when the input increased its value
Boolean	y	Output that is true when the input changes its value

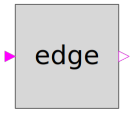
7.5.40 Edge

Output y is true, if the input u has a rising edge (y = edge(u))

Description: Block which outputs true if the Boolean input has a rising edge from false to true. Otherwise, the output is false.

7.5.40.1 CDL.Logical.Edge

Symbol



Parameters

Data Types	Name	Default	Description
Boolean	pre_u_start	false	For simulation, this parameter is used as the start value of pre(u).

Inputs

Data Types	Name	Description
Boolean	u	Input to be monitored

Outputs

Data Type	Name	Description
Boolean	y	Outputs true when the input switches to true

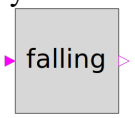
7.5.41 FallingEdge

Output y is true, if the input u has a falling edge (y = edge(not u))

Description: Block which outputs true if the Boolean input has a falling edge from true to false. Otherwise, the output is false.

7.5.41.1 CDL.Logical.FallingEdge

Symbol



Parameters

Data Types	Name	De-fault	Description
Boolean	pre_u_start	false	For simulation, this parameter is used as the start value of pre(u).

Inputs

Data Types	Name	Description
Boolean	u	Input to be monitored

Outputs

Data Type	Name	Description
Boolean	y	Outputs true when the input switches to false

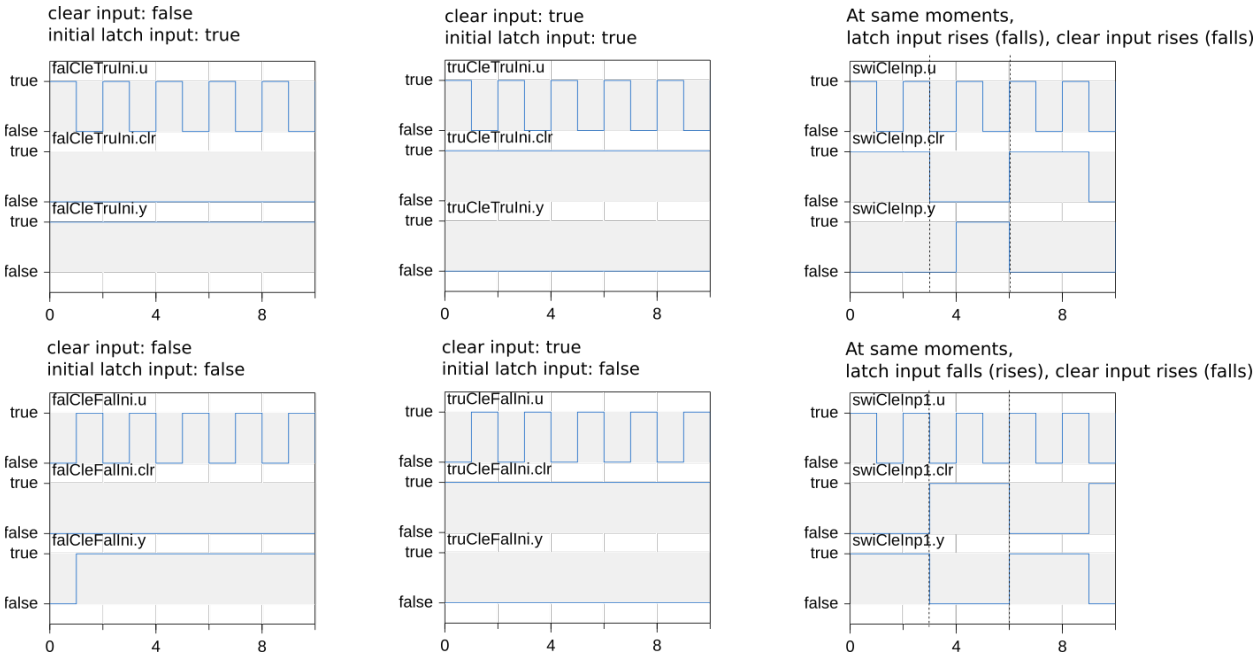
7.5.42 Latch

Maintains true until cleared.

Description: Block that generates a true output when the latch input u rises from false to true, provided that the clear input clr is false or also became at the same time false. The output remains true until the clear input clr rises from false to true.

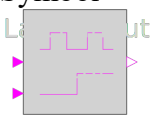
If the clear input clr is true, the output y switches to false (if it was true) and it remains false, regardless of the value of the latch input u.

At initial time, if clr = false, then the output will be y = u. Otherwise, it will be y=false (because the clear input clr is true).



7.5.42.1 CDL.Logical.Latch

Symbol



Parameters
N/A

Inputs

Data Types	Name	Description
Boolean	clr	Clear input
Boolean	u	Latch input

Outputs

Data Type	Name	Description
Boolean	y	Output with latched value

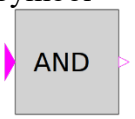
7.5.43 MultiAnd

Logical MultiAnd, y = u[1] and u[2] and u[3] and ...

Description: Block which outputs y = true if and only if all elements of the input vector u are true. If no connection to the input connector u is present, the output is y=false.

7.5.43.1 CDL.Logical.MultiAnd

Symbol



Parameters

Data Types	Name	De-fault	Description
Integer	nin	N/A	Number of input connections

Inputs

Data Types	Name	Description
Boolean	u[nin]	Inputs

Outputs

Data Type	Name	Description
Boolean	y	Output with true if all inputs are true

7.5.44 MultiOr

Elementary Block Names

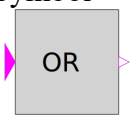
CDL	CXF
CDL.Logical.MultiOr	CXF.Logical.MultiOr

Logical MultiOr, y = u[1] or u[2] or u[3] or ...

Description: Block which outputs y=true if any element of the input vector u is true. If no connection to the input connector u is present, the output is y=false.

7.5.44.1 CDL.Logical.MultiOr

Symbol



Parameters

Data Types	Name	Default	Description
Integer	nin	N/A	Number of input connections

Inputs

Data Types	Name	Description
Boolean	u[nin]	Inputs

Outputs

Data Type	Name	Description
Boolean	y	Output with true if at least one input is true

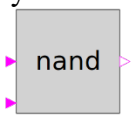
7.5.45 Nand

Logical 'nand': y = not (u1 and u2)

Description: Block which outputs true if at least one input is false. Otherwise, the output is false.

7.5.45.1 CDL.Logical.Nand

Symbol



Parameters
N/A

Inputs

Data Types	Name	Description
Boolean	u1	Input1 for 'nand'
Boolean	u2	Input2 for 'nand'

Outputs

Data Type	Name	Description
Boolean	y	Output with false if both inputs are true

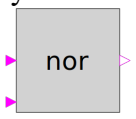
7.5.46 Nor

Logical 'nor': y = not (u1 or u2)

Description: Block which outputs true if none of the inputs is true. Otherwise, the output is false.

7.5.46.1 CDL.Logical.Nor

Symbol



Parameters
N/A

Inputs

Data Types	Name	Description
Boolean	u1	Input1 for 'nor'
Boolean	u2	Input2 for 'nor'

Outputs

Data Type	Name	Description

Boolean	y	Output with false if at least one of the inputs is true

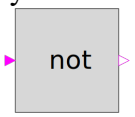
7.5.47 Not

Logical not

Description: Block which outputs true if the input is false, and false if the input is true.

7.5.47.1 CDL.Logical.Not

Symbol



Parameters
N/A

Inputs

Data Types	Name	Description
Boolean	u	Input to be negated

Outputs

Data Type	Name	Description
Boolean	y	Output with negated input

7.5.48 Or

Elementary Block Names

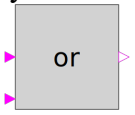
CDL	CXF
CDL.Logical.Or	CXF.Logical.Or

Logical 'or': y = u1 or u2

Description: Block which outputs true if at least one input is true. Otherwise, the output is false.

7.5.48.1 CDL.Logical.Or

Symbol



Parameters
N/A

Inputs

Data Types	Name	Description
Boolean	u1	Input1 for logical 'or'
Boolean	u2	Input2 for logical 'or'

Outputs

Data Type	Name	Description
Boolean	y	Output with true if at least one of the inputs is true

7.5.49 Proof

Verify two boolean inputs

Description: Block that compares a boolean set point u_s with a measured signal u_m and produces two outputs that may be used to raise alarms about malfunctioning equipment.

The block sets the output yLocFal = true if the set point is u_s = true but the measured signal is locked at false, i.e., u_m = false. Similarly, the block sets the output yLocTru = true if the set point is u_s = false but the measured signal is locked at true, i.e., u_m = true.

Informational Notes:

To use this block, proceed as follows: Set the parameter feedbackDelay ≥ 0 to specify how long the feedback of the controlled device is allowed to take to report its measured operational signal u_s after a set point change u_m. Set the parameter debounce ≥ 0 to specify how long the measured signal u_m need to remain constant for it to be considered stable. Connect the inputs for the set point u_s and the measured signal u_m to the output signals that need to be checked. If either output is true, raise an alarm, by connecting to the outputs of this block.

Any output being true indicates a problem.

The block has two timers that each start whenever the corresponding input changes. One timer, called feedbackDelay+debounce timer, starts whenever the set point u_s changes, and it runs for a time equal to feedbackDelay+debounce. The other timer, called debounce timer, starts whenever the measured signal u_m changes, and it runs for a time equal to debounce. The block starts verifying the inputs whenever the feedbackDelay+debounce timer lapsed, or the debounce timer lapsed, (and hence the measurement is stable,) whichever is first.

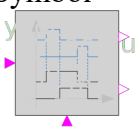
Both outputs being true indicates that the measured signal u_m is not stable within feedbackDelay+debounce time. Exactly one output being true indicates that the measured signal u_m is stable, but u_s ≠ u_m. In this case, the block sets yLocFal = true if u_s = true (the measured signal is locked at false), or it sets yLocTru = true if u_s = false (the measured signal is locked at true).

Therefore, exactly one output being true can be interpreted as follows: Suppose true means on and false means off.

Then, yLocTru = true indicates that an equipment is locked in operation mode but is commanded off; and similarly, yLocFal = true indicates that it is locked in off mode when it is commanded on.

7.5.49.1 CDL.Logical.Proof

Symbol



Parameters

Data Types	Name	Default	Description
Real	debounce		Time during which input must remain un-changed for signal to be considered valid and used in checks [seconds]
Real	feedbackDelay		Delay after which the two inputs are checked for equality once they become valid [seconds]

Inputs

Data Types	Name	Description
Boolean	u_m	Measured status
Boolean	u_s	Commanded status setpoint

Outputs

Data Types	Name	Description
Boolean	yLocFal	Output with true if the measured input is locked to false even after the setpoint has changed to true
Boolean	yLocTru	Output with true if the measured input is locked to true even after the setpoint has changed to false

7.5.50 Switch

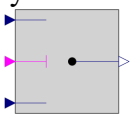
Switch between two inputs

Description: Block which outputs one of two input signals based on a Boolean input signal.

If the input signal u2 is true, the block outputs y = u1. Otherwise, it outputs y = u3.

7.5.50.1 CDL.Reals.Switch

Symbol



Parameters
N/A

Inputs

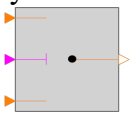
Data Types	Name	Description
Real	u1	Input u1
Boolean	u2	Boolean switch input signal, if true, y=u1, else y=u3
Real	u3	Input u3

Output

Data Type	Name	Description
Real	y	Output with u1 if u2 is true, else u3

7.5.50.2 CDL.Integers.Switch

Symbol



Parameters
N/A

Inputs

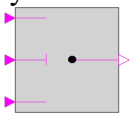
Data Types	Name	Description
Integer	u1	Input u1
Boolean	u2	Boolean switch input signal, if true, y=u1, else y=u3
Integer	u3	Input u3

Output

Data Type	Name	Description
Integer	y	Output with u1 if u2 is true, else u3

7.5.50.3 CDL.Logical.Switch

Symbol



Parameters
N/A

Inputs

Data Types	Name	Description
Boolean	u1	Input u1
Boolean	u2	Boolean switch input signal, if true, y=u1, else y=u3
Boolean	u3	Input u3

Output

Data Type	Name	Description
Boolean	y	Output with u1 if u2 is true, else u3

7.5.51 Toggle

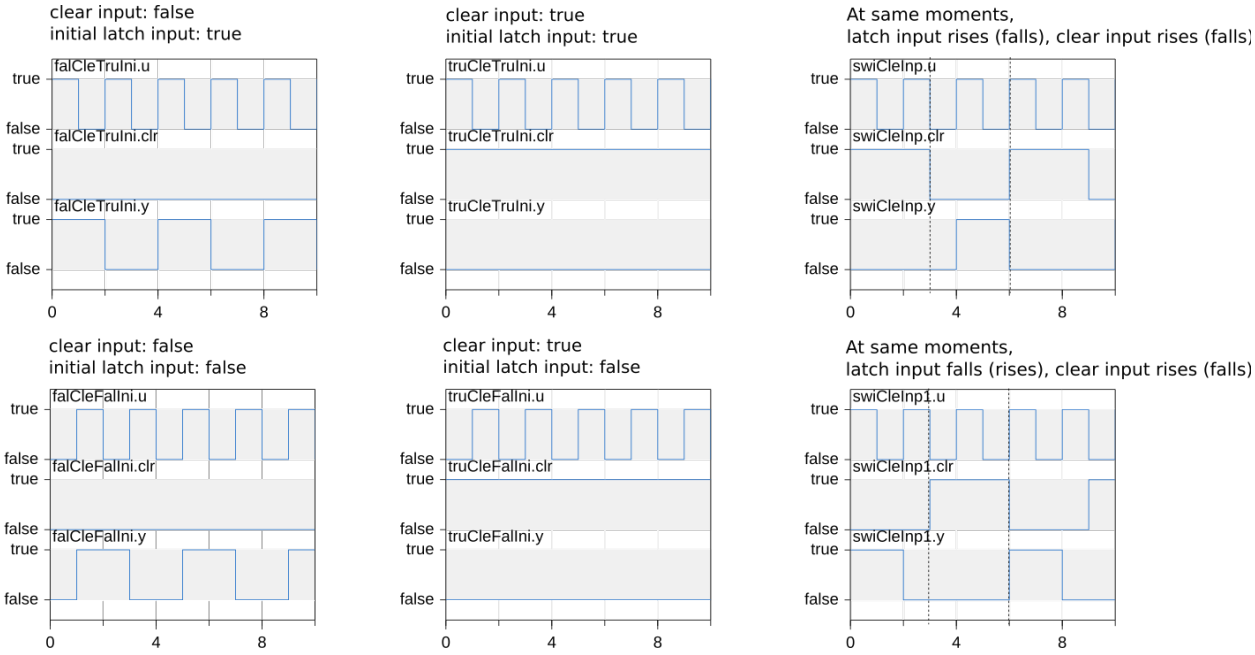
Toggles output value whenever its input turns true

Description: Block that generates a true output when toggle input u rises from false to true, provided that the clear input clr is false or also became at the same time false. The output remains true until

- the toggle input u rises from false to true again, or
- the clear input clr rises from false to true.

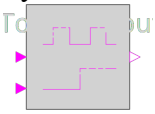
If the clear input clr is true, the output y switches to false (if it was true) and it remains false, regardless of the value of the toggle input u.

At initial time, if clr = false, then the output will be y = u. Otherwise, it will be y=false (because the clear input clr is true).



7.5.51.1 CDL.Logical.Toggle

Symbol



Parameters
N/A

Inputs

Data Types	Name	Description
Boolean	clr	Clear input
Boolean	u	Toggle Input

Outputs

Data Type	Name	Description
Boolean	y	Output with toggled signal

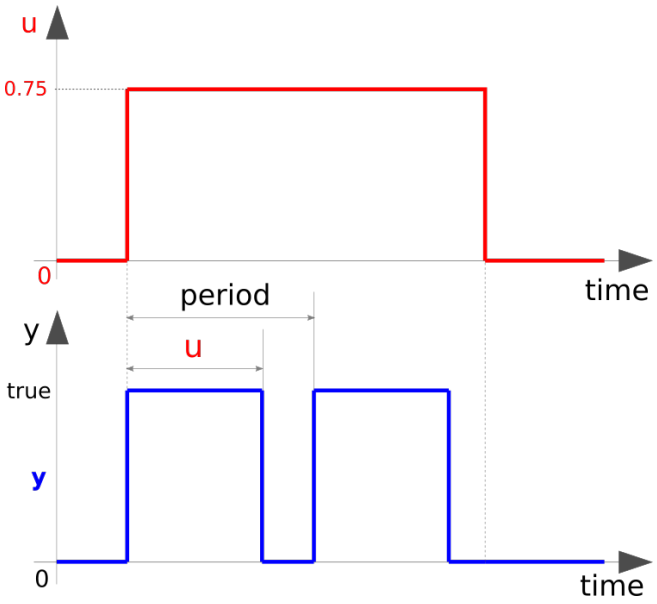
7.5.52 VariablePulse

Generate boolean pulse with the width specified by input

Description: The output of this block is a pulse with a constant period and a width as obtained from the input $0 \le u \le 1$, which is the width relative to the period.
The block produces the following outputs:

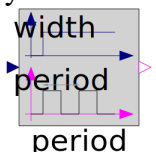
- If u = 0, the output y remains false.
- If 0 < u < 1, the output y will be a boolean pulse with the period specified by the parameter period and the width set to u*period.
- If u = 1, the output y remains true.

When the input u changes by more than deltaU and the output has been holding constant for more than minimum holding time minTruFalHol, the output will change to a new pulse with width equal to u*period.



7.5.52.1 CDL.Logical.VariablePulse

Symbol



Parameters

Data Type	Name	Default	Description
Real	deltaU	.01	Increment of u that triggers re-computation of output
Real	minTruFalHol	.01*period	Minimum time to hold true or false [s]
Real	period		Time for one pulse period [s]

Inputs

Data Type	Name	Description
Real	u	Ratio of the period that the output should be true [true=1]

Outputs

Data Type	Name	Description
Boolean	y	Boolean pulse when input is greater than zero

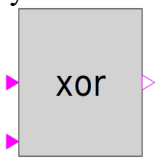
7.5.53 Xor

Logical 'xor': y = u1 xor u2

Description: Block which outputs true if exactly one input is true. Otherwise, the output is false.

7.5.53.1 CDL.Logical.Xor

Symbol



Parameters
N/A

Inputs

Data Type	Name	Description
Boolean	u1	Input1 for logical 'xor'
Boolean	u2	Input2 for logical 'xor'

Outputs

Data Type	Name	Description
Boolean	y	Output with u1 xor u2

7.5.54 DewPoint_TDryBulPhi

Block to compute the dew point temperature based on relative humidity.

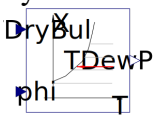
Description:

Dew point temperature calculation for moist air above freezing temperature.

The correlation used in this block is valid for dew point temperatures between *0*°C and *93*°C. It is the correlation from 2009 ASHRAE Handbook Fundamentals, p. 1.9, Equation 39.

7.5.54.1 CDL.Psychrometrics.DewPoint_TDryBulPhi

Symbol



Parameters
N/A

Inputs

Data Type	Name	Description
Real	phi	Relative Humidity
Real	TDryBul	Dry bulb temperature

Outputs

Data Type	Name	Description
Real	TDewPoi	Dewpoint Temperature

7.5.55 SpecificEnthalpy_TDryBulPhi

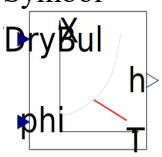
Block to compute the specific enthalpy based on relative humidity

Description:

This block computes the specific enthalpy for a given dry bulb temperature, relative air humidity, and atmospheric pressure. The specific enthalpy is zero if the temperature is *0*°C and if there is no The correlation used in this model is from the 2009 ASHRAE Handbook Fundamentals, p. 1.9, Equation 32.

7.5.55.1 CDL.Psychrometrics.SpecificEnthalpy_TDryBulPhi

Symbol



Parameters

Data Type	Name	Default	Description
Real	pAtm	101325	Atmospheric pressure [pascals]

Inputs

Data Type	Name	Description
Real	phi	Relative Humidity
Real	TDryBul	Dry bulb temperature

Outputs

Data Type	Name	Description
Real	h	Specific Enthalpy

7.5.56 WetBulb_TDryBulPhi

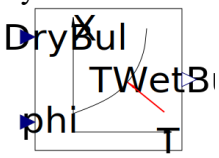
Block to compute the wet bulb temperature based on relative humidity.

Description:

This block computes the wet bulb temperature for a given dry bulb temperature, relative air humidity, and atmospheric pressure. See appendix for links to reference code.

7.5.56.1 CDL.Psychrometrics.WetBulb_TDryBulPhi

Symbol



Parameters
N/A

Inputs

Data Type	Name	Description
Real	TDryBul	Dry bulb temperature
Real	phi	Relative humidity

Outputs

Data Type	Name	Description

Real	TWetBul	Wet bulb temperature

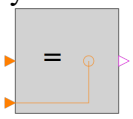
7.5.57 Equal

Output y is true if input u1 is equal to input u2

Description: Block which outputs true if the input u1 is equal to the input u2. Otherwise, the output is false.

7.5.57.1 CDL.Integers.Equal

Symbol



Parameters
N/A

Inputs

Data Type	Name	Description
Integer	u1	Input to be checked for equality with other input
Integer	u2	Input to be checked for equality with other input

Outputs

Data Type	Name	Description
Boolean	y	Outputs that is true if the two inputs are equal, and false otherwise

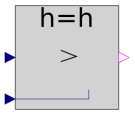
7.5.58 Greater

Output y is true, if input u1 is greater than input u2

Description: Block which outputs true if the input u1 is greater than input u2. Otherwise, the output is false.

7.5.58.1 CDL.Reals.Greater

Symbol



Parameters

Data Type	Name	Default	Description
Real	h	0	Hysteresis
Boolean	pre_y_start	false	Value of pre(y) at initial time

Note: The parameter h ≥ 0 is used to specify a hysteresis. For any $h \geq 0$, the output switches to true if $u_1 > u_2$, and it switches to false if $u_1 \leq u_2 - h$. Note that in the special case of $h = 0$, this produces the output $y=u_1 > u_2$.
To disable hysteresis, set h=0.

Inputs

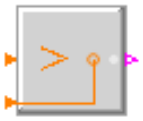
Data Type	Name	Description
Real	u1	First input u1
Real	u2	Second input u2

Outputs

Data Type	Name	Description
Boolean	y	Outputs true if u1 is greater than u2

7.5.58.2 CDL.Integers.Greater

Symbol



Parameters
N/A

Inputs

Data Type	Name	Description
Integer	u1	First input u1
Integer	u2	Second input u2

Outputs

Data Type	Name	Description
Boolean	y	Outputs true if u1 is greater than u2

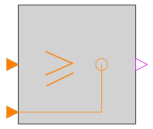
7.5.59 GreaterEqual

Output y is true, if input u1 is greater than or equal to input u2.

Description: Block which outputs true if the input u1 is equal to or greater than the input u2. Otherwise, the output is false.

7.5.59.1 CDL.Integers.GreaterEqual

Symbol



Parameters
N/A

Inputs

Data Type	Name	Description
Integer	U1	First input u1
Integer	u2	Second input u2

Outputs

Data Type	Name	Description
Boolean	y	Outputs true if u1 is greater or equal than u2

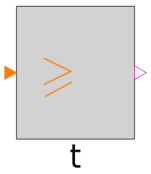
7.5.60 GreaterEqualThreshold

Output y is true, if input u1 is greater than or equal to threshold.

Description: Block which outputs true if the input is greater than or equal to the parameter t. Otherwise, the output is false.

7.5.60.1 CDL.Integers.GreaterEqualThreshold

Symbol



Parameters

Data Types	Name	Default	Description
Integer	t	0	Threshold against which the input is compared to

Inputs

Data Type	Name	Description
Integer	u	Input to be compared against the threshold

Outputs

Data Type	Name	Description
Boolean	y	Outputs true if u is greater or equal than the threshold

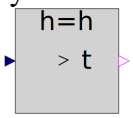
7.5.61 GreaterThreshold

Output y is true, if input u is greater than a threshold

Description: Block which outputs true if the input is greater than the parameter t. Otherwise, the output is false.

7.5.61.1 CDL.Reals.GreaterThreshold

Symbol:



Parameters

Data Types	Name	Default	Description
Real	h	0	Hysteresis
Boolean	pre_y_start	false	Value of pre(y) at initial time
Real	t	0	Threshold comparison

Note: The parameter h ≥ 0 is used to specify a hysteresis. For any $h \geq 0$, the output switches to true if $u > t$, where t is the threshold, and it switches to false if $u \leq t - h$. Note that in the special case of $h = 0$, this produces the output $y=u > t$.

To disable hysteresis, set h=0.

Inputs

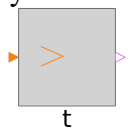
Data Type	Name	Description
Real	u	Input to be compared against the threshold

Outputs

Data Type	Name	Description
Boolean	y	Outputs true if u is greater than the threshold with hysteresis

7.5.61.2 CDL.Integers.GreaterThreshold

Symbol:



Parameters

Data Types	Name	Default	Description
Integer	t	0	Threshold comparison

Inputs

Data Type	Name	Description
Integer	u	Input to be compared against the threshold

Outputs

Data Type	Name	Description
Boolean	y	Outputs true if u is greater than the threshold with hysteresis

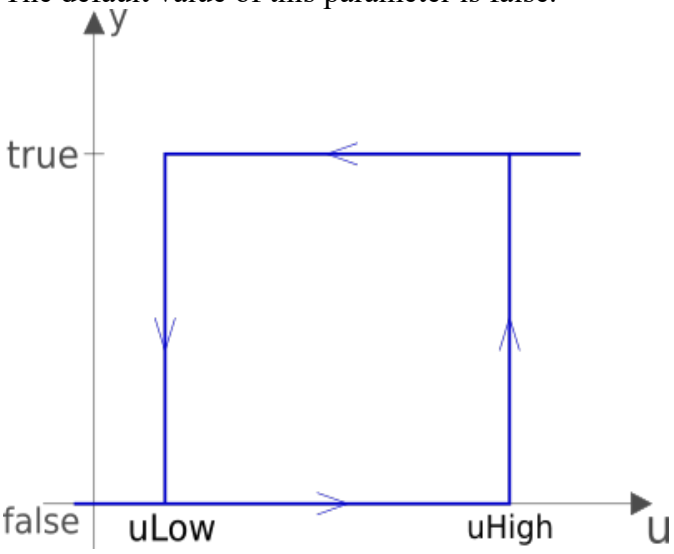
7.5.62 Hysteresis

Transform Real to Boolean signal with Hysteresis.

Description: Block that transforms a Real input signal into a Boolean output signal:

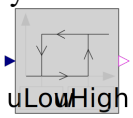
- When the output was false and the input becomes greater than the parameter uHigh, the output switches to true.
- When the output was true and the input becomes less than the parameter uLow, the output switches to false.

The start value of the output is defined via parameter pre_y_start (= value of pre(y) at initial time). The default value of this parameter is false.



7.5.62.1 CDL.Reals.Hysteresis

Symbol



Parameters:

Data Type	Name	Default	Description
Boolean	pre_y_start	false	Value of pre(y) used at initial time
Real	uHigh	0	If y=false and u>uHigh, switch to y=true
Real	uLow	0	If y=true and u<uLow, switch to y=false

Inputs

Data Type	Name	Description
Real	u	Input to be compared against hysteresis values

Outputs

Data Type	Name	Description
Boolean	y	Output value of comparison

7.5.63 Less

Output y is true, if input u1 is less than input u2

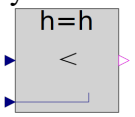
Description: Block which outputs true if the input u1 is less than input u2. The Real or Analog version of this block has an optional hysteresis h. Hysteresis does not apply for the Integer version of this block.

The parameter $h \geq 0$ is used to specify a hysteresis. If $h \neq 0$, then the output switches to true if $u_1 > u_2$, and it switches to false if $u_1 < u_2 - h$. If $h = 0$, the output is $y = u_1 > u_2$.

Informational Note:
Enabling hysteresis can avoid frequent switching. Adding hysteresis is recommended in real controllers to guard against sensor noise, and in simulation, to guard against numerical noise. Numerical noise can be present if an input depends on a state variable or a quantity that requires an iterative solution, such as a temperature or a mass flow rate of an HVAC system. To disable hysteresis, set h=0.

7.5.63.1 CDL.Reals.Less

Symbol



Parameters

Data Type	Name	Default	Description
Real	h	0	Hysteresis
Real	Pre_y_start	false	Value of pre(y) used at initial time

Inputs

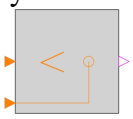
Data Type	Name	Description
Real	u1	First input u1
Real	u2	Second input u2

Outputs

Data Type	Name	Description
Boolean	y	Outputs true if u1 is less than u2

7.5.63.2 CDL.Integers.Less

Symbol



Parameters
N/A

Inputs

Data Type	Name	Description
Integer	u1	First input u1
Integer	u2	Second input u2

Outputs

Data Type	Name	Description
Boolean	y	Outputs true if u1 is less than u2

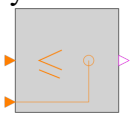
7.5.64 LessEqual

Output y is true, if input u1 is less than or equal to input u2

Description: Block outputs `true` if the input `u1` is equal to or less than the input `u2`. Otherwise, the output is `false`

7.5.64.1 CDL.Integers.LessEqual

Symbol



Parameters
N/A

Inputs

Data Type	Name	Description
Integer	U1	First input
Integer	u2	Second input

Outputs

Data Type	Name	Description

Boolean	y	Outputs true if u1 is less or equal than u2

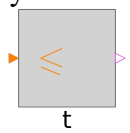
7.5.65 LessEqualThreshold

Output y is true, if input u is less than or equal to threshold

Description: Block which outputs true if the input is less than or equal to the parameter t. Otherwise, the output is false.

7.5.65.1 CDL.Integers.LessEqualThreshold

Symbol



Parameters

Data Type	Name	Default	Description
Integer	t	0	Threshold for comparison

Inputs

Data Types	Name	Description
Integer	u	Input to be compared

Outputs

Data Type	Name	Description
Boolean	y	Outputs true if u is less or equal than the threshold

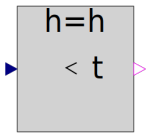
7.5.66 LessThreshold

Output y is true, if input u is less than a threshold.

Description: Block which outputs true if the input is less than the parameter t. Otherwise, the output is false.

7.5.66.1 CDL.Reals.LessEqualThreshold

Symbol:



Parameters

Data Type	Name	Default	Description
Real	t	0	Threshold for comparison
Real	h	0	Hysteresis

Note: The parameter h ≥ 0 is used to specify a hysteresis. For any $h \geq 0$, the output switches to true if $u < t$, where t is the threshold, and it switches to false if $u \geq t + h$. Note that in the special case of $h = 0$, this produces the output $y=u < t$.

Inputs

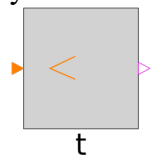
Data Type	Name	Description
Real	u	Input to be compared against the threshold

Outputs

Data Type	Name	Description
Boolean	y	Outputs true if u is less than the threshold with hysteresis

7.5.66.2 CDL.Integers.LessEqualThreshold

Symbol:



Parameters

Data Type	Name	Default	Description
Integer	t	0	Threshold for comparison

Inputs

Data Type	Name	Description

Integer	u	Input to be compared against the threshold

Outputs

Data Type	Name	Description
Boolean	y	Outputs true if u is less than the threshold

7.5.67 Limiter

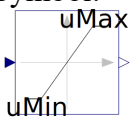
Limit the range of an input

Description: Block that outputs y = min(uMax, max(uMin, u)), where u is an input and uMax and uMin are parameters.

If uMax < uMin, an error occurs.

7.5.67.1 CDL.Reals.Limiter

Symbol:



Parameters

Data Types	Name	Default	Description
Real	uMax	N/A	Upper limit of input
Real	uMin	N/A	Lower limit of input

Inputs

Data Type	Name	Description
Real	u	Input to be limited

Outputs

Data Type	Name	Description
Real	y	Limited value of input

7.5.68 OnCounter

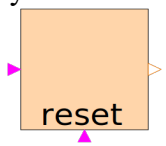
Increment the output if the input switches to true

Description: Block which outputs how often the trigger input changed to true since the last invocation of reset.

This block may be used as a counter. Its output y starts at the parameter value y_start. Whenever the input signal trigger changes to true, the output is incremented by *1*. When the input reset changes to true, then the output is reset to y = y_start. If both inputs trigger and reset change simultaneously, then the output is y = y_start.

7.5.68.1 CDL.Integers.OnCounter

Symbol:



Parameters

Data Type	Name	Default	Description
Integer	y_start	0	Initial and reset value of y if input reset switches to true

Inputs

Data Type	Name	Description
Boolean	reset	Reset, when true, the counter is set to y_start
Boolean	trigger	Trigger, when set to true, the counter increases

Outputs

Data Type	Name	Description
Integer	y	Counter value

7.5.69 Pre

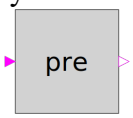
Breaks algebraic loops by an infinitesimal small-time delay (y = pre(u): event iteration continues until u = pre(u)). Typically used in simulation and not in control.

Description: This block delays the Boolean input by an infinitesimal small-time delay and therefore, breaks algebraic loops. In a network of logical blocks, in every *closed connection loop*, at least one logical block must have a delay, since algebraic systems of Boolean equations are not solvable.

This block returns the value of the input signal u from the last event iteration. The event iteration stops once both values are identical, i.e., if u = pre(u).

7.5.69.1 CDL.Logical.Pre

Symbol:



Parameters

Data Type	Name	Default	Description
Boolean	pre_u_start	false	Start value of pre(u) at initial time

Inputs

Data Type	Name	Description
Boolean	u	Input to be delayed by one event iteration

Outputs

Data Type	Name	Description
Boolean	y	Input delayed by one event iteration

7.5.70 Ramp

Limit the changing rate of the input.

Description:
Block that limits the rate of change of the input u by a ramp if the boolean input active is true, otherwise the block outputs y = u.

This block computes a threshold for the rate of change between input u and output y as thr = (u-y)/Td, where Td > 0 is a parameter. The output y is computed as follows:

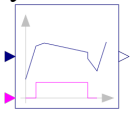
- If thr < fallingSlewRate, then dy/dt = fallingSlewRate,
- if thr > raisingSlewRate, then dy/dt = raisingSlewRate,
- otherwise, dy/dt = thr.

A smaller time constant Td >0 means a higher accuracy for the derivative approximation.

Note that when the input switches to false, the output y can have a discontinuity.

7.5.70.1 CDL.Reals.Ramp

Symbol



Parameters

Data Type	Name	Default	Description
Real	fallingSlewRate	-raisingSlewRate	Maximum speed with which to decrease the output [1/s]
Real	raisingSlewRate		Maximum speed with which to increase the output [1/s]
Real	Td	raisingSlewRate*0.001	Derivative time constant

Inputs

Data Type	Name	Description
Boolean	active	Set to true to enable rate limiter
Real	u	Input that is being rate limited

Outpus

Data Type	Name	Description
Real	y	Rate limited output if active is true, else output is equal to input

7.5.71 Stage

Output total stages that should be enabled

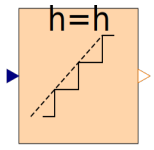
Description: Block which outputs the total number of stages to be enabled.

The block compares the input u with the threshold of each stage. If the input is greater than a stage threshold, the output is set to that stage. The block outputs the total number of stages to be enabled.

The parameter n specifies the maximum number of stages. The stage thresholds are evenly distributed, i.e., the thresholds for stages [1, 2, 3, ... , n] are [0, 1/n, 2/n, ... , (n-1)/n], plus a hysteresis which is by default h=0.02/n. Once the output changes, it cannot change for at least holdDuration seconds.

7.5.71.1 CDL.Reals.Stage

Symbol:



Parameters

Data Types	Name	Default	Description

Real	h	0.02/n	Hysteresis for comparing input with threshold
Real	holdDuration		Minimum time that the output needs to be held constant. Set to 0 to disable hold time [s]
Integer	n		Number of stages that could be enabled
Integer	pre_y_start	0	Value of pre(y) at initial time

Inputs

Data Type	Name	Description
Real	u	Input between 0 and 1 for specifying stages

Outputs

Data Type	Name	Description
Integer	y	Total number of stages that should be enabled

7.5.72 BooleanToInteger

Converts a boolean to an integer signal

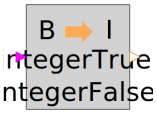
Description: Block that outputs the Integer equivalent of the Boolean input.

```
y = if u then integerTrue else integerFalse;
```

where u is of Boolean and y of Integer type, and integerTrue and integerFalse are parameters.

7.5.72.1 CDL.Conversions.BooleanToInteger

Symbol:



Parameters

Data Type	Name	Default	Description
Integer	integerFalse	0	Output signal for false Boolean
Integer	integerTrue	1	Output signal for true Boolean

Inputs

Data Type	Name	Description
Boolean	u	Boolean to be converted to an Integer

Outputs

Data Type	Name	Description
Integer	y	Converted input as an Integer

7.5.73 BooleanToReal

Converts a boolean to a real

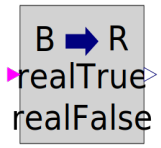
Description: Block which outputs the Real equivalent of the Boolean input.

```
y = if u then realTrue else realFalse;
```

where u is of Boolean and y of Real type, and realTrue and realFalse are parameters.

7.5.73.1 CDL.Conversions.BooleanToReal

Symbol:



Parameters

Data Type	Name	Default	Description
Real	realFalse	0	Output for false Boolean Input
Real	realTrue	1	Output for true Boolean Input

Inputs

Data Type	Name	Description
Boolean	u	Boolean to be converted to a Real

Outputs

Data Type	Name	Description
Real	y	Converted input as a Real

7.5.74 IntegerToReal

Converts an integer to a real signal

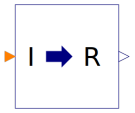
Description: Block that outputs the Real equivalent of the Integer input.

```
y = u;
```

where u is of Integer and y of Real type.

7.5.74.1 CDL.Conversions.IntegerToReal

Symbol:



Parameters
N/A

Inputs

Data Type	Name	Description
Integer	u	Integer to be converted to a Real

Outputs

Data Type	Name	Description
Real	y	Converted input as a Real

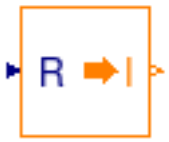
7.5.75 RealToInteger

Converts a real to an integer signal

Description Block that outputs y as the nearest integer value of the input u.
The block outputs
`y = integer( floor( u + 0.5 ) )  if u > 0,  y = integer( ceil ( u - 0.5 ) )  otherwise.`

7.5.75.1 CDL.Conversions.RealToInteger

Symbol:



Parameters
N/A

Inputs

Data Type	Name	Description

Real	u	Real to be converted to an Integer

Outputs

Data Type	Name	Description
Integer	y	Converted input as an Integer

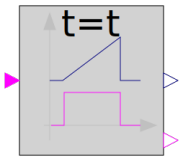
7.5.76 Timer

Timer measuring the time from the time instant where the Boolean input became true

Description: If the Boolean input u is true, the output y is the time that has elapsed since u became true. Otherwise, y is *0*. If the output y becomes greater than the threshold time t, the output passed is true. Otherwise it is false.
In the limiting case where the timer value reaches the threshold t and the input u becomes false simultaneously, the output passed remains false.

7.5.76.1 CDL.Logical.Timer

Symbol:



Parameters

Data Type	Name	Default	Description
Real	t	0	Threshold time for comparison [s]

Inputs

Data Type	Name	Description
Boolean	u	Input that switches timer on if true, and off if false

Outputs

Data Type	Name	Description
Boolean	passed	Output with true if the elapsed time is greater than threshold
Real	y	Elapsed time in seconds

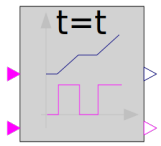
7.5.77 TimerAccumulating

Accumulating timer that can be reset

Description: Timer that accumulates time until it is reset by an input signal.
If the Boolean input u is true, the output y is the time that has elapsed while u has been true since the last time reset became true. If u is false, the output y holds its value. If the output y becomes greater than the threshold time t, the output passed is true. Otherwise, it is false.
When reset becomes true, the timer is reset to 0.
In the limiting case where the timer value reaches the threshold t and the input u becomes false simultaneously, the output passed remains false.

7.5.77.1 CDL.Logical.TimerAccumulating

Symbol:



Parameters

Data Type	Name	Default	Description
Real	t	0	Threshold time for comparison [s]

Inputs

Data Type	Name	Description
Boolean	reset	Input for signal that sets timer to zero if it switches to true
Boolean	u	Input that switches timer on if true, and off if false

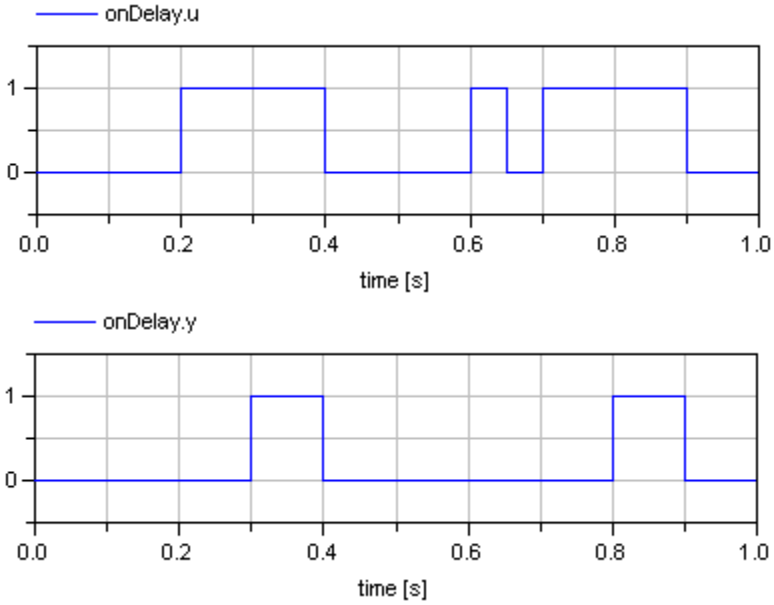
Outputs

Data Type	Name	Description
Boolean	passed	Output with true if the elapsed time is greater than threshold
Real	y	Elapsed time [s]

7.5.78 TrueDelay

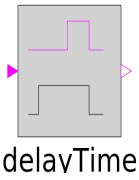
Delay a rising edge of the input, but do not delay a falling edge.

Description: Block which delays a signal when it becomes true.
A rising edge of the Boolean input u gives a delayed output. A falling edge of the input is immediately given to the output. If delayOnInit = true, then a true input signal at the start time is also delayed, otherwise, the input signal is produced immediately at the output.
Simulation results of a typical example with a delay time of *0.1* second is shown below.



7.5.78.1 CDL.Logical.TrueDelay

Symbol:



Parameters

Data Type	Name	Default	Description
Boolean	delayOnInit	false	Set to true to delay initial true input
Real	delayTime	0	Delay time [s]

Inputs

Data Type	Name	Description
Boolean	u	Input to be delayed when it switches to true

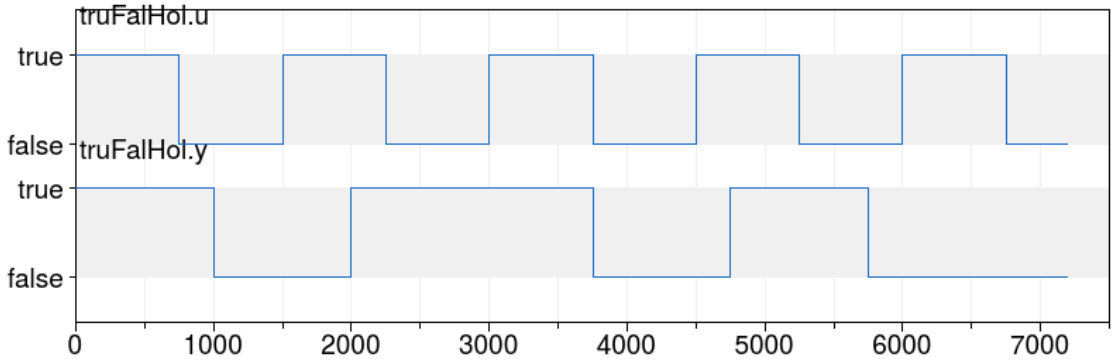
Outputs

Data Type	Name	Description
Boolean	y	Output with delayed input after it switched to true

7.5.79 TrueFalseHold

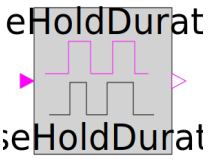
When logical input changes state the logical output matches the state and remains in that state for a minimum delay parameter.

Description: Block which holds a true or false signal for at least a defined time period.
Whenever the input u switches to true (resp. false), the output y switches and remains true for at least the duration specified by the parameter trueHoldDuration (resp. falseHoldDuration). After this duration has elapsed, the output will be y = u.
This block could for example be used to disable an economizer, and not re-enable it for *10* min, and vice versa.
Simulation results of a typical example with trueHoldDuration = falseHoldDuration = 1000 s.



7.5.79.1 CDL.Logical.TrueFalseHold

Symbol:



Parameters

Data Type	Name	Default	Description
Real	falseHoldDura-tion	trueHoldDuration	Duration of false hold

Real	trueHoldDura-tion	N/A	Duration of true hold

Inputs

Data Type	Name	Description
Boolean	u	Input that is to be delayed

Outputs

Data Type	Name	Description
Boolean	y	Output with delayed input

7.5.80 Sources.CalendarTime

Outputs the current time, day, month, and year.

Description: Outputs values related to time and date, including the current hour and minute in the day, day of the week (1-7), and the current month and day. Note that daylight savings time is not considered in this component.

7.5.80.1 CDL.Reals.Sources.CalendarTime

Symbol:

Parameters

Data Types	Name	Default	Description
Enumeration	zerTim		Enumeration for choosing how time 0 is selected (see Section 7.7)
Integer	yearRef	2016	Year when time=0 if zerTim=cus-tom
Advanced			
Real	offset	0	For calculating time in different Time Zones

Inputs
NA

Outputs

Data Type	Name	Description
Integer	year	Year
Integer	month	Month of the year
Integer	day	Day of the month
Integer	hour	Hour of the day
Real	minute	Minute of the hour
Integer	weekday	Day of the week (Monday=1, Sunday=7)

Informational Note:

The data type for Minute is Real while the data types for the other outputs are Integer. This is to allow for faster calculations during simulations.

7.5.81 Sources.CivilTime

Outputs the current system or simulation time.

Description: This block outputs the time of the model or in the case of a building automation system, the building automation system synchronized time, and hence need to assign a value for the output of this block. Daylight saving time shall not be considered, e.g. the block always outputs civil time rather than daylight savings time.

If a simulation starts at $t=-1$, then this block outputs first $t=-1$, and its output is advanced at the same rate as the simulation time.

7.5.81.1 CDL.Reals.Sources.CivilTime

Symbol:

Parameters
N/A

Inputs
NA

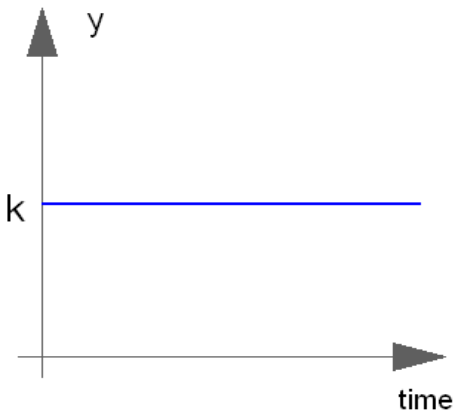
Outputs

Data Type	Name	Description
Real	Y	Civil Time [s]

7.5.82 Sources.Constant

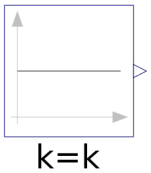
Outputs a constant value.

Description: Block that outputs a constant signal `y = k`, where `k` is a valued parameter that can be type Real, Integer, or Boolean.



7.5.82.1 CDL.Reals.Sources.Constant

Symbol:



Parameters

Data Type	Name	Default	Description
Real	k	n/a	Constant value

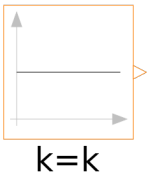
Inputs
NA

Outputs

Data Type	Name	Description
Real	y	Constant value output equal to parameter k

7.5.82.2 CDL.Integers.Sources.Constant

Symbol:



Parameters

Data Type	Name	Default	Description
Integer	k	n/a	Constant value

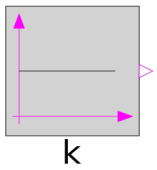
Inputs
NA

Outputs

Data Type	Name	Description
Integer	y	Constant value output equal to parameter k

7.5.82.3 CDL.Logicals.Sources.Constant

Symbol:



Parameters

Data Type	Name	Default	Description

Boolean	k	n/a	Constant value

Inputs
NA

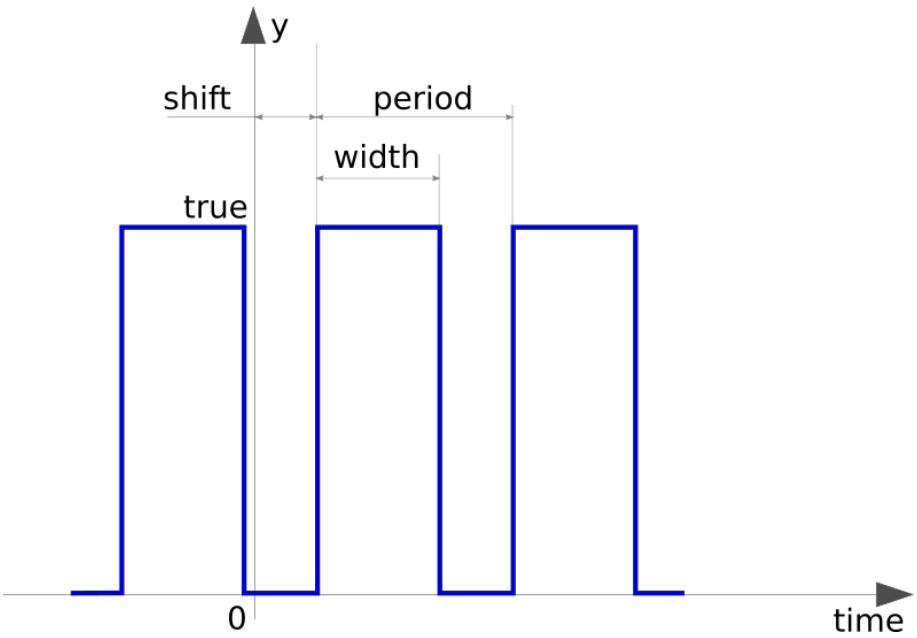
Outputs

Data Type	Name	Description
Boolean	y	Constant value output equal to parameter k

7.5.83 Sources.Pulse

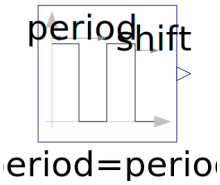
Outputs a pulse at a specific frequency and duration.

Description: Provides a pulse output with a specific period and width.



7.5.83.1 CDL.Reals.Sources.Pulse

Symbol:



Parameters

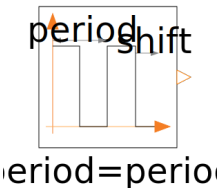
Data Types	Name	Default	Description
Real	amplitude	1.0, 1	Amplitude of the pulse
Real	offset	0.0, 0	Offset of the output
Real	period	n/a	Time for one period in seconds
Real	shift		Shift time for outputs in seconds
Real	width	.5	Width of period as a fraction of 1

Inputs
NA

Data Type	Name	Description
Real, Integer, Boolean	y	Pulse output

7.5.83.2 CDL.Integers.Sources.Pulse

Symbol:



Parameters

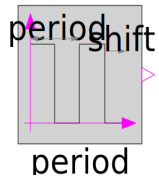
Data Types	Name	Default	Description
Integer	amplitude	1.0, 1	Amplitude of the pulse
Integer	offset	0.0, 0	Offset of the output
Real	period	n/a	Time for one period in seconds
Real	shift		Shift time for outputs in seconds
Real	width	.5	Width of period as a fraction of 1

Inputs
NA

Data Type	Name	Description
Integer	y	Pulse output

7.5.83.3 CDL.Logicals.Sources.Pulse

Symbol:



Parameters

Data Types	Name	Default	Description
Real	period	n/a	Time for one period in seconds
Real	shift		Shift time for outputs in seconds
Real	width	.5	Width of period as a fraction of 1

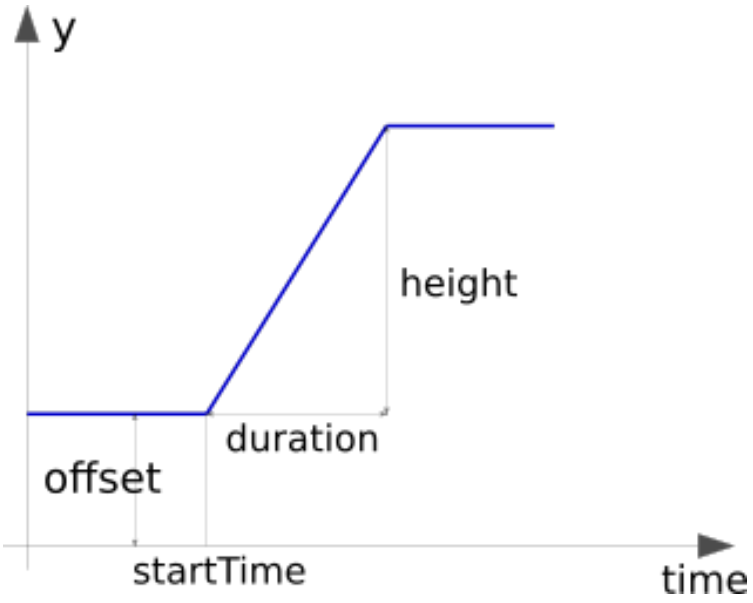
Inputs
NA

Data Type	Name	Description
Boolean	y	Pulse output

7.5.84 Sources.Ramp

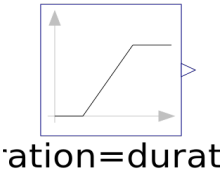
Generates a ramp signal of specified height and duration.

Description: Outputs a Real value based on the parameters for height, duration, start time, and offset.



7.5.84.1 CDL.Reals.Sources.Ramp

Symbol:



Parameters

Data Type	Name	Default	Description
Real	duration	n/a	Duration of ramp in seconds
Real	height	1	Height of ramp
Real	offset	0	Offset of ramp
Real	startTime	0	Output = offset for time < start time in seconds

Inputs
NA

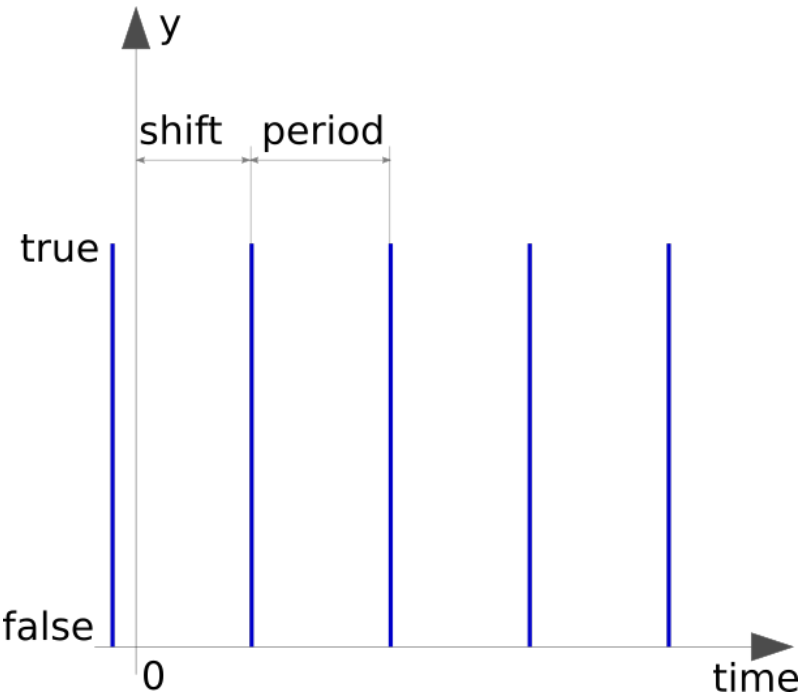
Outputs

Data Type	Name	Description
Real	y	Ramp output value

7.5.85 Sources.SampleTrigger

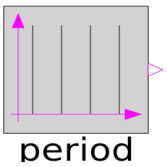
Outputs a sample trigger

Description: The Boolean output y is a trigger that is only true at sample times (defined by parameter period) and is otherwise false.



7.5.85.1 CDL.Logical.Sources.SampleTrigger

Symbol:



Parameters

Data Types	Name	Default	Description
Real	period		Sample period in seconds

Real	shift	0	Shift time for output in seconds

Inputs
NA

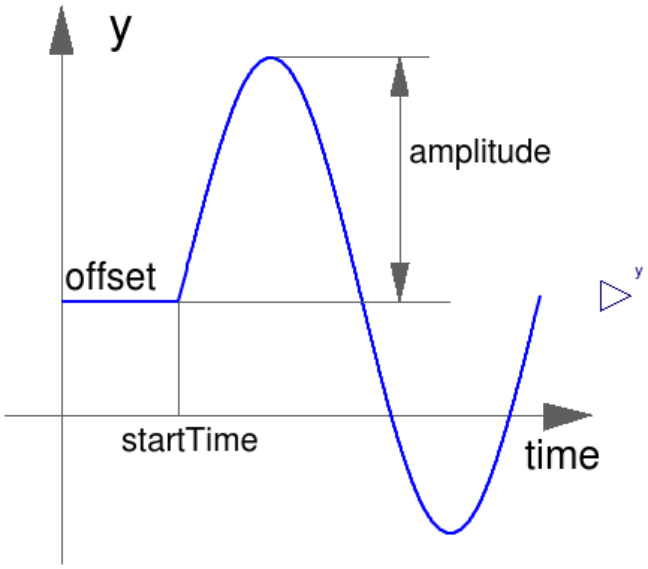
Outputs

Data Type	Name	Description
Boolean	y	Output with trigger value

7.5.86 Sources.Sin

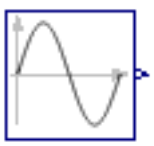
Generate a sine wave

Description: Outputs a sine wave based on the parameters for amplitude, offset, time, and start time.



7.5.86.1 CDL.Reals.Sources.Sin

Symbol:



Parameters

Data Types	Name	Default	Description
Real	amplitude	1	Amplitude of sine wave
Real	offset	0	Offset of output signal in Seconds
Real	freqHz		Frequency of sine wave in Hz.
Real	phase	0	Phase of sine wave in Radians
Real	startTime	0	Output=offset for time < start-Time in seconds

Inputs
NA

Outputs

Data Type	Name	Description
Real	y	Sine output signal

7.5.87 Sources.TimeTable

A listing of times – used for elementary scheduling

Description: Block which outputs time table values.
The block takes as a parameter a timetable of a format:

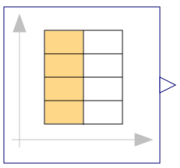
```
table = [ 0*3600 , 0;
        6*3600, 1;
        6*3600, 0;
       18*3600, 1;
       18*3600, 1];
period = 24*3600;
```

where the first column of `table` is time and the remaining column(s) are the table values. The time column contains `Real` values that are in units of seconds if `timeScale = 1`. The parameter `timeScale` can be used to scale the time values, for example, use `timeScale = 3600` if the values in the first column are interpreted as hours.

The values in column two and higher must be `0` or `1`, otherwise, the model stops with an error.
Until a new tabulated value is set, the previous tabulated value is returned.
The table scope is repeated periodically with periodicity `period`.

7.5.87.1 CDL.Reals.Sources.TimeTable

Symbol:



Parameters

Data Types	Name	Default	Description
Real	period		Periodicity of table in seconds
Real	table[:,:]		Table matrix with time as a first table column (in seconds, unless timeScale is not 1) and Integers in all other columns
Real	timeScale	1	Time scale – set to 60 for minutes, 3,600 for hours (in seconds)

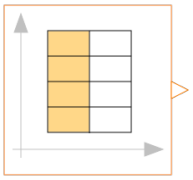
Inputs
NA

Outputs

Data Type	Name	Description
Real, Integer, Boolean	y	Output with tabulated values

7.5.87.2 CDL.Integers.Sources.TimeTable

Symbol:



Parameters

Data Types	Name	Default	Description
Real	period		Periodicity of table in seconds
Real	table[:,:]		Table matrix with time as a first table column (in seconds, unless

			timeScale is not 1) and Integers in all other columns
Real	timeScale	1	Time scale – set to 60 for minutes, 3,600 for hours (in seconds)

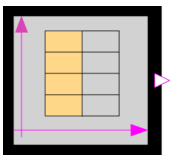
Inputs
NA

Outputs

Data Type	Name	Description
Integer	y	Output with tabulated values

7.5.87.3 CDL.Logical.Sources.TimeTable

Symbol:



Parameters

Data Types	Name	Default	Description
Real	period		Periodicity of table in seconds
Real	table[:,:]		Table matrix with time as a first table column (in seconds, unless timeScale is not 1) and Integers in all other columns
Real	timeScale	1	Time scale – set to 60 for minutes, 3,600 for hours (in seconds)

Inputs
NA

Outputs

Data Type	Name	Description
Boolean	y	Output with tabulated values

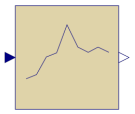
7.5.88 FirstOrderHold

First order hold of a sampled data system

Description: Block that outputs the extrapolation through the values of the last two sampled input signals.

7.5.88.1 CDL.Discrete.FirstOrderHold

Symbol:



Parameters

Data Type	Name	Default	Description
Real	samplePeriod		Sample period of component [seconds]

Inputs

Data Type	Name	Description
Real	u	Input to be sampled

Outputs

Data Type	Name	Description
Real	y	First order hold of input

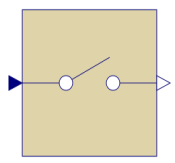
7.5.89 Sampler

Ideal sampler of a continuous input signal

Description: Block that outputs the input, sampled at a sampling rate defined via parameter samplePeriod.

7.5.89.1 CDL.Discrete.Sampler

Symbol:



Parameters

Data Type	Name	Default	Description
Real	samplePeriod		Sample period of component [seconds]

Inputs

Data Type	Name	Description
Real	u	Input to be sampled

Outputs

Data Type	Name	Description
Real	y	Sampled input

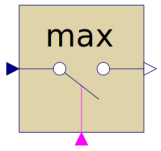
7.5.90 TriggeredMax

Output the maximum absolute value of an input at trigger instants.

Description Block that outputs the input whenever the trigger input is rising (i.e., trigger changes to true). The maximum, absolute value of the input at the sampling point is provided as the output.

7.5.90.1 CDL.Discrete.TriggeredMax

Symbol:



Parameters
N/A

Inputs

Data Type	Name	Description
Boolean	trigger	Input for trigger that causes u to be sampled
Real	u	Input to be sampled

Outputs

Data Type	Name	Description
Real	y	Maximum of input over all trigger instants

7.5.91 TriggeredMovingMean

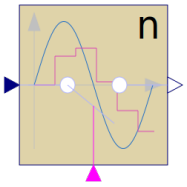
Triggered discrete moving mean of an input

Description: Block which outputs the triggered moving mean value of an input signal.

When the trigger signal is rising (i.e., the trigger changes to true), the block samples the input, computes the moving mean value over the past `n` samples, and produces this value at its output `y`.

7.5.91.1 CDL.Discrete.TriggeredMovingMean

Symbol:



Parameters

Data Types	Name	Default	Description
Integer	n		Number of samples over which the input is averaged

Inputs

Data Type	Name	Description
Boolean	trigger	Input to trigger ththat causes u to be sampled
Real	u	Input to be sampled

Outputs

Data Type	Name	Description
Real	y	Moving mean of input over all trigger in-stants

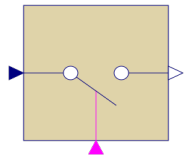
7.5.92 TriggeredSampler

Triggered sampling of an input

Description: Samples the input whenever the triggerinput is rising (i.e., trigger changes from false to true) and provides the sampled input as output. Before the first sampling, the output is equal to the initial value defined via parameter y_start.

7.5.92.1 CDL.Discrete.TriggeredSampler

Symbol:



Parameters

Data Type	Name	Default	Description
Real	y_start	0	Initial value of output

Inputs

Data Type	Name	Description
Boolean	trigger	Input for trigger that causes u to be sampled
Real	u	Input to be sampled

Outputs

Data Type	Name	Description
Real	y	Input at the last trigger instant

7.5.93 UnitDelay

Outputs the input with a unit delay

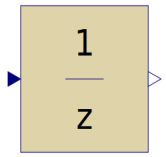
Description: Block that outputs the input signal with a unit delay:

$$y = \frac{1}{z} * u$$

Output y is the value of input u of the previous sample instant. Before the second sample instant, the output y is identical to parameter y_start.

7.5.93.1 CDL.Discrete.UnitDelay

Symbol:



Parameters

Data Types	Name	Default	Description
Real	samplePeriod		Sample period
Real	y_start	0	Initial value of output

Inputs

Data Type	Name	Description
Real	u	Input to be sampled

Outputs

Data Type	Name	Description
Real	y	Input at previous sample instant

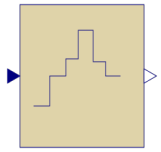
7.5.94 ZeroOrderHold

Outputs an input with a zero-order hold

Description: Block that outputs the sampled input at sample time instants. The output is held at the value of the last sample instant during the sample points. At initial time, the block feeds the input directly to the output.

7.5.94.1 CDL.Discrete.ZeroOrderHold

Symbol:



Parameters

Data Types	Name	Default	Description
Real	samplePeriod		Sample period

Inputs

Data Type	Name	Description
Real	u	Input to be sampled

Outputs

Data Type	Name	Description
Real	y	Zero order hold of the input

7.5.95 ExtractSignal
Extract signals from an input signal vector

Description: Extract signals from the vector-valued input signal and transfer them to the vector-valued output signal.
This vector specifies which input signals are taken and in which order they are transferred to the output vector. Note that the dimension of extract has to match the number of outputs and the elements of extract has to be in the range of [1, nin]. Additionally, the dimensions of the input connector signals and the output connector signals have to be explicitly defined via the parameters nin and nout.

Informational Note:

The specification

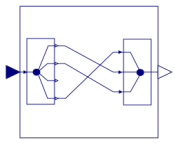
```
nin = 7 "Number of inputs";
nout = 4 "Number of outputs";
extract[nout] = {6,3,3,2} "Extracting vector";
```

extracts four output signals (nout=4) from the seven elements of the input vector (nin=7):

```
y[1, 2, 3, 4] = u[6, 3, 3, 2];
```

7.5.95.1 CDL.Routing.RealExtractSignal

Symbol:



‹tract=extra

Parameters

Data Types	Name	Default	Description
Real	extract[nout]	1:nout	Extracting vector
Integer	nin	1	Number of inputs
Integer	nout	1	Number of outputs

Inputs

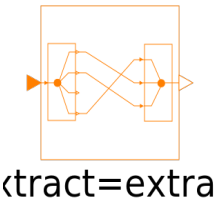
Data Type	Name	Description
Real	u[nin]	Input signals

Outputs

Data Type	Name	Description
Real	y[nout]	Signals extracted from the input vector with the extraction scheme specified by the integer vector

7.5.95.2 CDL.Routing.IntegerExtractSignal

Symbol:



Parameters

Data Types	Name	Default	Description
Real	extract[nout]	1:nout	Extracting vector
Integer	nin	1	Number of inputs
Integer	nout	1	Number of outputs

Inputs

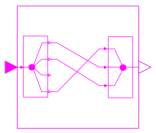
Data Type	Name	Description
Integer	u[nin]	Input signals

Outputs

Data Type	Name	Description
Integer	y[nout]	Signals extracted from the input vector with the extraction scheme specified by the integer vector

7.5.95.3 CDL.Routing.BooleanExtractSignal

Symbol:



‹tract=extra

Parameters

Data Types	Name	Default	Description
Real	extract[nout]	1:nout	Extracting vector
Integer	nin	1	Number of inputs
Integer	nout	1	Number of outputs

Inputs

Data Type	Name	Description
Boolean	u[nin]	Input signals

Outputs

Data Type	Name	Description
Boolean	y[nout]	Signals extracted from the input vector with the extraction scheme specified by the integer vector

7.5.96 Extractor

Extract scalar signals from an input vector depending on an output index

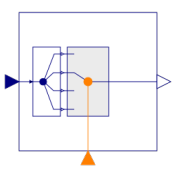
Description: Block that returns

```
y = u[index];
```

When the index is out of range, then y = u[nin] if index > nin, and y = u[1] if index < 1.

7.5.96.1 CDL.Routing.RealExtractor

Symbol:



Parameters

Data Type	Name	Default	Description
Integer	nin	1	Number of inputs

Inputs

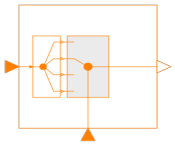
Data Type	Name	Description
Integer	index	Index of input vector elements to be extracted out
Real	u[nin]	Input signals to be extracted

Outputs

Data Type	Name	Description
Real	y	Signal extracted from the input vector

7.5.96.2 CDL.Routing.IntegerExtractor

Symbol:



Parameters

Data Type	Name	Default	Description
Integer	nin	1	Number of inputs

Inputs

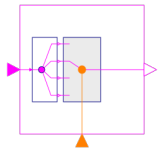
Data Type	Name	Description
Integer	index	Index of input vector elements to be extracted out
Integer	u[nin]	Input signals to be extracted

Outputs

Data Type	Name	Description
Integer	y	Signal extracted from the input vector

7.5.96.3 CDL.Routing.BooleanExtractor

Symbol:



Parameters

Data Type	Name	Default	Description
Integer	nin	1	Number of inputs

Inputs

Data Type	Name	Description
Integer	index	Index of input vector elements to be extracted out
Boolean	u[nin]	Input signals to be extracted

Outputs

Data Type	Name	Description
Boolean	y	Signal extracted from the input vector

7.5.97 ScalarReplicator

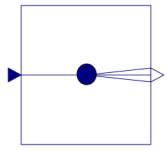
Elementary Block Names

CDL	**CXF**
CDL.Routing.BooleanScalarReplicator	CXF.Routing.BooleanScalarReplicator
CDL.Routing.IntegersScalarReplicator	CXF.Routing.IntegerScalarReplicator
CDL.Routing.RealScalarReplicator	CXF.Routing.RealScalarReplicator
	CXF.Routing.AnalogScalarReplicator

Scalar replicator
Description: This block replicates the input signal to an array of nout identical output signals.

7.5.97.1 CDL.Routing.RealScalarReplicator

Symbol:



Parameters

Data Types	Name	Default	Description
Integer	nout	1	Number of outputs

Inputs

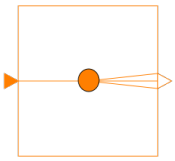
Data Type	Name	Description
Real	u	Input signals to be replicated

Outputs

Data Type	Name	Description
Real	y[nout]	Output with replicated input signal

7.5.97.2 CDL.Routing.IntegerScalarReplicator

Symbol:



Parameters

Data Types	Name	Default	Description
Integer	nout	1	Number of outputs

Inputs

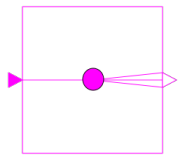
Data Type	Name	Description
Integer	u	Input signals to be replicated

Outputs

Data Type	Name	Description
Integer	y[nout]	Output with replicated input signal

7.5.97.3 CDL.Routing.BooleanScalarReplicator

Symbol:



Parameters

Data Types	Name	Default	Description
Integer	nout	1	Number of outputs

Inputs

Data Type	Name	Description
Boolean	u	Input signals to be replicated

Outputs

Data Type	Name	Description
Boolean	y[nout]	Output with replicated input signal

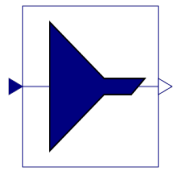
7.5.98 VectorFilter

Filter an input vector based on a boolean mask.

Description: This block filters a vector of size nin to a vector of size nout given a Boolean mask msk.
If an entry in msk is true, then the value of this input will be sent to the output y, otherwise, it will be discarded.
The parameter msk must have exactly nout entries set to true, otherwise an error message shall be issued.

7.5.98.1 CDL.Routing.RealVectorFilter

Symbol:



Parameters

Data Type	Name	Default	Description
Boolean	msk[nin]	fill(true,nin)	Array mask

Integer	nin		Size of input vector
Integer	nout		Size of output vector

Inputs

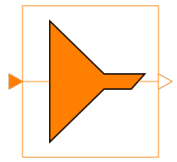
Data Type	Name	Description
Real	u[nin]	Input signals from which values are extracted

Outputs

Data Type	Name	Description
Real	y[nout]	Output with extracted input signals

7.5.98.2 CDL.Routing.IntegerVectorFilter

Symbol:



Parameters

Data Type	Name	Default	Description
Boolean	msk[nin]	fill(true,nin)	Array mask
Integer	nin		Size of input vector
Integer	nout		Size of output vector

Inputs

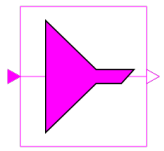
Data Type	Name	Description
Integer	u[nin]	Input signals from which values are extracted

Outputs

Data Type	Name	Description
Integer	y[nout]	Output with extracted input signals

7.5.98.3 CDL.Routing.BooleanVectorFilter

Symbol:



Parameters

Data Type	Name	Default	Description
Boolean	msk[nin]	fill(true,nin)	Array mask
Integer	nin		Size of input vector
Integer	nout		Size of output vector

Inputs

Data Type	Name	Description
Boolean, Integer, Real	u[nin]	Input signals from which values are extracted

Outputs

Data Type	Name	Description
Boolean, Integer, Real	y[nout]	Output with extracted input signals

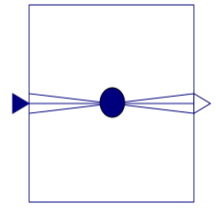
7.5.99 VectorReplicator

Vector signal replicator

Description: This block replicates a vector input signal of size nin, to a matrix with nout rows and nin columns, where each row is duplicating the input vector.

7.5.99.1 CDL.Routing.RealVectorReplicator

Symbol:



Parameters

Data Types	Name	Default	Description
Integer	nin	1	Size of input vector

Integer	nout	1	Number of rows in output

Inputs

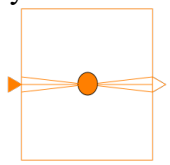
Data Type	Name	Description
Real	u[nin]	Vector input signals to be replicated

Outputs

Data Type	Name	Description
Real	y[nout,nin]	Output with replicated input signals

7.5.99.2 CDL.Routing.IntegerVectorReplicator

Symbol:



Parameters

Data Types	Name	Default	Description
Integer	nin	1	Size of input vector
Integer	nout	1	Number of rows in output

Inputs

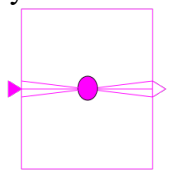
Data Type	Name	Description
Integer	u[nin]	Vector input signals to be replicated

Outputs

Data Type	Name	Description
Integer	y[nout,nin]	Output with replicated input signals

7.5.99.3 CDL.Routing.BooleanVectorReplicator

Symbol:



Parameters

Data Types	Name	Default	Description
Integer	nin	1	Size of input vector
Integer	nout	1	Number of rows in output

Inputs

Data Type	Name	Description
Boolean	u[nin]	Vector input signals to be replicated

Outputs

Data Type	Name	Description
Boolean	y[nout,nin]	Output with replicated input signals

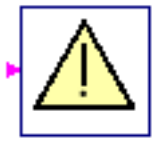
7.5.100 Assert

Block which writes a message when an input becomes false

Description: Tools or control systems shall write a message together with a time stamp to an output device and/or a log file.

7.5.100.1CDL.Utilities.Assert

Symbol:



Parameters

Data Type	Name	Default	Description
String	message		Message written when u becomes false

Inputs

Data Type	Name	Description
Boolean	u	Input that triggers the assert when its value is false.

Outputs
N/A

7.5.101SunRiseSet

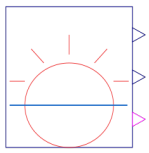
Outputs the time for sunrise and sunset.

Description: Outputs the next sunrise and sunset time. The sunrise time keeps constant until the next sunrise, at which time the output gets updated. Similarly, the output for the next sunset is updated at each sunset.

The time zone parameter is based on UTC time. Note that Eastern Standard Time is UTC-5 hours. This block only supports US standard time and not daylight savings time.

7.5.101.1CDL.Utilities.SunRiseSet

Symbol:



Parameters
It has the following parameters

Data Types	Name	Default	Description
Real	lat		Latitude (radians)
Real	lon		Longitude (radians)
Real	timZon		Time Zone (hours)

Inputs
NA

Outputs

Data Type	Name	Description
Real	nextSunRise	Time of next sunrise in seconds
Real	nextSunSet	Time of next sunset in seconds
Boolean	sunUp	Output True if the sun is up

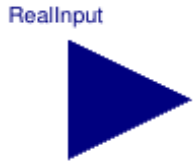
7.5.102 InputConnectors

Connector with an input of a specific data type

Description: Provides an input

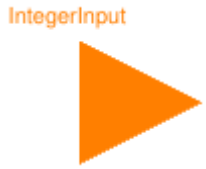
7.5.102.1CDL.Interfaces.RealInput

Symbol:



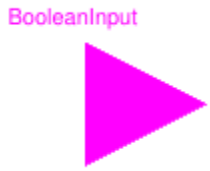
7.5.102.2 CDL.Interfaces.IntegerInput

Symbol:



7.5.102.3CDL.Interfaces.BooleanInput

Symbol:



7.5.103 OutputConnectors

Connector with one output of a specific data type

Description: Provides an output

7.5.103.1CDL.Interfaces.RealOutput

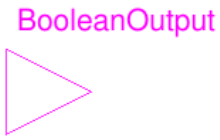
Symbol:

7.5.103.2 CDL.Interfaces.IntegerOutput

Symbol:

7.5.103.3CDL.Interfaces.BooleanOutput

Symbol:



7.6 Predefined constants

The standard also defines the following immutable constants in the `CDL.Constants` package:

```
  constant Real eps=1E-15
    "Biggest number such that 1.0 + eps = 1.0";
  constant Real small=1E-37
    "Smallest number such that small and -small
are representable on the machine";
  constant Real pi=2*Modelica.Math.asin(1.0)
    "Constant number pi, 3.14159265358979";
```

Informal note: The constants `eps` and `small` are typically used to restrict the minimum value of parameters that need to be bigger than zero, such as a control gain, in a consistent way.

7.7 Predefined enumerations

CDL also contains the following types in the package CDL.Types.

Informational note: Types are used to declare Integer-valued parameter, restrict their possible values, and associate a human-understandable value with the parameter. For example, for a PID controller, rather than allowing a configuration 1, 2, 3, and 4, using types allows to set the configuration to P, PI, PD, or PID.

The following types are defined in CDL.Types:

```
type Extrapolation = enumeration(
    HoldLastPoint
  "Hold the first/last table point outside of the table scope",
    LastTwoPoints
  "Extrapolate by using the derivative at the first/last table points outside of the table scope",
    Periodic
  "Repeat the table scope periodically")
  "Enumeration defining the extrapolation of time table interpolation";
type SimpleController = enumeration(
    P
  "P controller",
    PI
  "PI controller",
    PD
  "PD controller",
    PID
  "PID controller")
  "Enumeration defining P, PI, PD, or PID simple controller type";
type Smoothness = enumeration(
    LinearSegments
  "Table points are linearly interpolated",
    ConstantSegments
  "Table points are not interpolated, but the previous tabulated value is returned")
  "Enumeration defining the smoothness of table interpolation";
```

```
type ZeroTime = enumeration(
    UnixTimeStamp "Thu, 01 Jan 1970 00:00:00 local time",
    UnixTimeStampGMT "Thu, 01 Jan 1970 00:00:00 GMT",
    Custom "User specified local time",
    NY2010 "New year 2010, 00:00:00 local time",
    NY2011 "New year 2011, 00:00:00 local time",
    NY2012 "New year 2012, 00:00:00 local time",
    NY2013 "New year 2013, 00:00:00 local time",
    NY2014 "New year 2014, 00:00:00 local time",
    NY2015 "New year 2015, 00:00:00 local time",
    NY2016 "New year 2016, 00:00:00 local time",
    NY2017 "New year 2017, 00:00:00 local time",
    NY2018 "New year 2018, 00:00:00 local time",
    NY2019 "New year 2019, 00:00:00 local time",
    NY2020 "New year 2020, 00:00:00 local time",
    NY2021 "New year 2021, 00:00:00 local time",
    NY2022 "New year 2022, 00:00:00 local time",
    NY2023 "New year 2023, 00:00:00 local time",
    NY2024 "New year 2024, 00:00:00 local time",
    NY2025 "New year 2025, 00:00:00 local time",
    NY2026 "New year 2026, 00:00:00 local time",
    NY2027 "New year 2027, 00:00:00 local time",
    NY2028 "New year 2028, 00:00:00 local time",
    NY2029 "New year 2029, 00:00:00 local time",
    NY2030 "New year 2030, 00:00:00 local time",
    NY2031 "New year 2031, 00:00:00 local time",
    NY2032 "New year 2032, 00:00:00 local time",
    NY2033 "New year 2033, 00:00:00 local time",
    NY2034 "New year 2034, 00:00:00 local time",
    NY2035 "New year 2035, 00:00:00 local time",
    NY2036 "New year 2036, 00:00:00 local time",
```

```
    NY2037 "New year 2037, 00:00:00 local time",
    NY2038 "New year 2038, 00:00:00 local time",
    NY2039 "New year 2039, 00:00:00 local time",
    NY2040 "New year 2040, 00:00:00 local time",
    NY2041 "New year 2041, 00:00:00 local time",
    NY2042 "New year 2042, 00:00:00 local time",
    NY2043 "New year 2043, 00:00:00 local time",
    NY2044 "New year 2044, 00:00:00 local time",
    NY2045 "New year 2045, 00:00:00 local time",
    NY2046 "New year 2046, 00:00:00 local time",
    NY2047 "New year 2047, 00:00:00 local time",
    NY2048 "New year 2048, 00:00:00 local time",
    NY2049 "New year 2049, 00:00:00 local time",
    NY2050 "New year 2050, 00:00:00 local time")
  "Use this to set the date corresponding to time = 0 in CDL.Reals.Sources.CalendarTime";
```

8 APPENDICES

(The appendices are not part of this standard. It is merely informative and does not contain requirements necessary for conformance to the standard. It has not been processed according to the ANSI requirements for a standard and may contain material that has not been subject to public review or a consensus process. Unresolved objectors on informative material are not offered the right to appeal at ASHRAE or ANSI.)

8.1 – Overview of Standard

8.1.1 Document Structure

This document defines a Controls Description Language which is intended to allow for designers, modelers, control contractors and integrators, commissioning agents, and owners to have a standard methodology for describing and sharing the building control system logic. The focus is on the specific logic that is programmed into the control system. Examples of this logic would include the control sequence of an air handler, or the coordination needed to properly reset the pump static pressure in the chiller plant included in ASHRAE Guideline 36. It is important to note that most commercially available control systems also contain other logic and processing, which while it could be documented using this standard, is not specifically covered in this standard.

Specific items covered in this standard include:

- The definition of the Controls Description Language (CDL) and the Controls Exchange Format (CXF), including the syntax which allows for composing and documenting control logic.
- Definitions for Elementary Functions and Elementary Blocks, as well as composite and extension blocks.

These allow for the following:

- Ability to define a library of sequences which can be applied based on the selected options for a project to define an instance of a sequence.
- Models created in CDL can be used with Modelica tools to test the logic. Closed loop control simulations can also be performed using a Modelica model of the HVAC system.
- Converting (or translating) from CDL to CXF.
- The import of CDL and CXF into tools used to define programming for control systems.
- The movement of control logic from one control system to another.
- The conversion (or translation) from CXF back to CDL – noting there may be some manual decision-making required.

The following functions are **not** covered or included in this standard and may be covered in other ASHRAE documents or by proprietary methods:

- Network communications.

- Alarm and event processing and notification.
- Time of day scheduling and calendars.
- Trending and reporting.
- Fault detection diagnostics and analytics (unless the rules are part of the control logic).
- Preprocessing of input signals to digital conversion, and scaling.
- Post processing of logical outputs and connection to actuators.
- Other control logic not documented in CDL or CXF

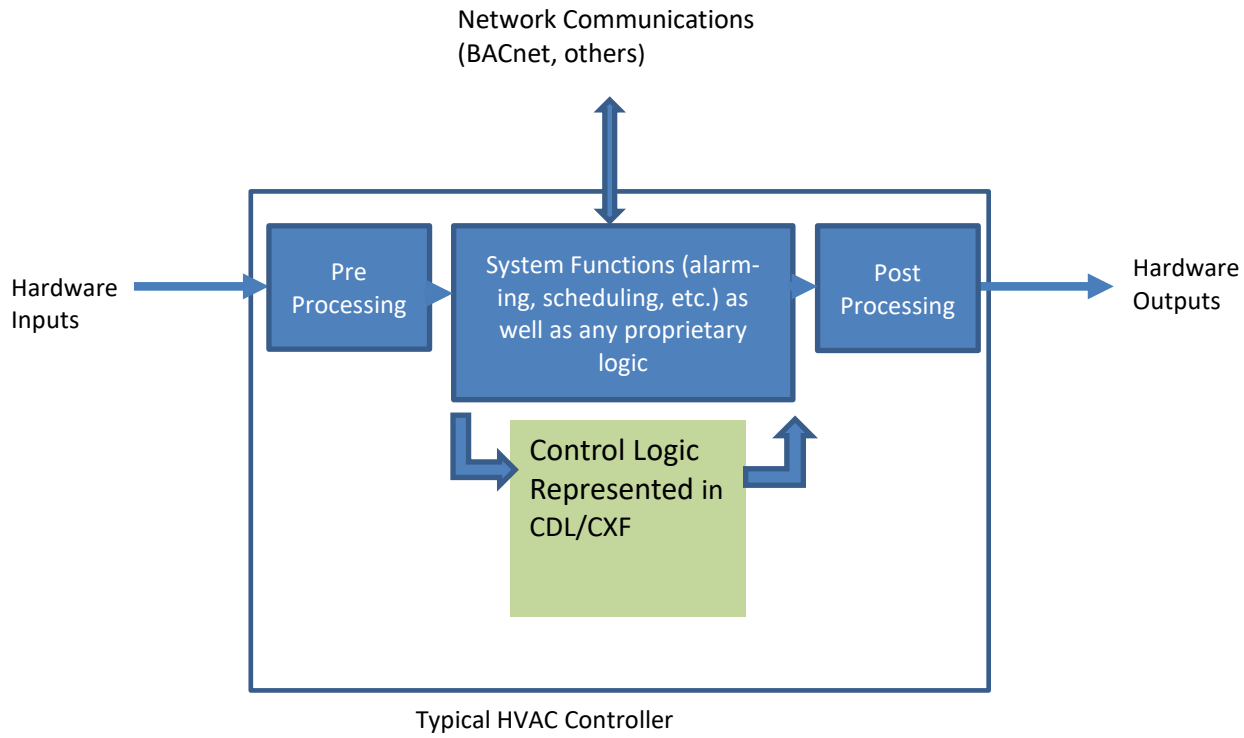


Figure 8-1: Diagram represents functions in a typical commercially available HVAC controller. The control logic shown in the bottom box is covered in this standard. All other logic is defined by the control system provider – and could use CDL.

The document starts with sections that define the tile, purpose, and scope followed by Sections 5, 6, and 7 which contain the primary technical content. Sections 5 and 6 define the language that is used to describe the inputs, outputs, connectors, and Elementary Blocks, and how to compose control logic using Composite Blocks. These are referred to as the CDL and the CXF. Section 7 defines the Elementary Blocks which are represented in CDL and CXF and define the low-level mathematical operations such as adding two inputs of a block and producing at its output the sum of these inputs.

CDL:
The CDL format is intended to be used during the process of designing a control system which includes defining the control logic and then testing performance in whole building simulation.

CDL is defined using an open standard modeling language called Modelica[7]. CDL files can readily be used with various applications that support Modelica to allow the control logic to be verified and tested. CDL also can be used as part of an energy modeling simulation (or co-simulation to evaluate the energy performance of a sequence with a specific model). For example, energy simulation using CDL is being supported by the United States Department of Energy's Modelica Buildings Library and Spawn of EnergyPlus.[8]

CXF:
The CXF format is intended to be used primarily in the deployment of control logic as part of the delivery of a control system. The CXF file is defined using an internet standard (ECMA-404) called Java Script Object Notation (JSON).[9] The intent of the CXF file is that it will be supported by control suppliers for use by controls contractors and integrators as a format to import or export control logic. This means that tools that are used to visualize, edit, error check, upload, and download control logic would be compatible with this standard. There are several potential uses for CXF. These include:

- Import from Design: The US Department of Energy has developed libraries of control sequences in CDL. This includes sequences from ASHRAE Guideline 36 as well as other high-performance sequences. These sequences can be machine translated from CDL to CXF and be provided to the controls contractor or integrator as a representation of the desired control logic.
- Export from a Control System: Existing control logic may be stored in the CXF format. Once it is in this interoperable format it can be readily used for import into another control system, or it can also be machine translated into the CDL format and used for simulations, modeling, and the development of ongoing modeling and comparison tools.

Elementary Blocks:
To define a control description language, there are several elements needed. The most basic element is called an "Elementary Block." Elementary Blocks range from basic math (add, equal, absolute value) to logic (if then, else), to special and control functions (sunrise / sunset, PID). To use elementary blocks, one instantiates these blocks, assigns their parameters (such as a proportional gain), and connects their inputs and outputs. CDL and CXF both contain elementary blocks that provide the same mathematical functionality, thereby allowing translation from CDL to CXF. See Section 7.3 for more details.

"Composite Blocks" are a collection of any number of Elementary Blocks and other Composite Blocks. Composite Block can declare parameters (such as for the sampling time that may be used by two blocks inside this Composite Block), Connectors for inputs and outputs, and how these Connectors are connected.

Elementary Blocks have a defined set of inputs and outputs, as well as adjustable or configurable parameters. Specific logic, which is defined mathematically, occurs within each block. For example, a block could be used to add the values from two inputs and output the result. Note that both

[7] https://modelica.org
[8] https://www.energy.gov/eere/buildings/downloads/spawn-energyplus-spawn
[9] https://www.json.org/json-en.html

CDL and CXF use the same elementary functions, and the two representations contain essentially the same information but are intended to be used by different applications.

To define a control logic, instances of Elementary Blocks are connected to each other or to instances of Composite Blocks, so that the output of one block becomes the input of another block. Logic can be viewed visually as a series of blocks with connecting lines showing these relationships. The graphical representation can also be defined in a machine-readable file, which includes the definitions for all the blocks and how they are connected.

System Functions Outside of CDL:
Commercially available control systems often include support for various "system" functions. Examples of this include scheduling, alarm processing, trending, communications processing, and processing to support connected I/O. See Figure A-1.

At a minimum, this standard requires that vendors shall be able to readily interchange data from their system functions to and from the logic contained in CDL. A CDL sequence will have a series of inputs and outputs that provide this connection. The definition of how this information is connected and the tools to support this connection are up to the vendor and are not defined in this standard.

Extension Blocks:
While this standard defines a specific set of Elementary Blocks, there will be cases in which a new block is required that cannot be specified using a Composite Block (for example, because it requires execution of advanced mathematics that is beyond this standard). These new blocks are referred to in this document as extension blocks and are defined in Section 5.

Proprietary Control Logic:
The purpose of this standard is to encourage the open sharing of control logic. This can readily be done with the use of Elementary, Composite, and Extension blocks. But it is also recognized that there are cases in which a vendor may not elect to share logic in an open and interoperable manner. There are two methods to keep control logic as proprietary. The first is to utilize an Extension block which documents all required inputs, outputs, and parameters, but the logic would not be documented. The second approach is to hold the proprietary logic as part of the "System Functions" of the controller.

8.1.2 Control Sequence Libraries and Instances

When a designer selects HVAC equipment for a project, they will typically select the options that are needed for the project, from a list of potential options available from suppliers. As an example, consider a multi-zone air handler. The designer will typically make selections for how the unit will deal with outdoor air, the location and types of coils, the fan configuration, etc. What is finally specified and shown in the project schedule is the specific configuration (or instance) of each air handler required for a specific project. So, the designer started with a library of all options for air handlers, and selected the specific instance required for the project.

The same concept can be used for control design – see Figure 8.1. CDL can have a library of potential control options and based on a series of decisions, there can be a specific instance of a control sequence which is useable for the specific equipment options selected. The use of libraries

offers several benefits, including the ability to reuse code and assure consistency. To support libraries, CDL includes support for functionalities such as matrixes, vectors, and the assignment of parameter values using functions. Many control systems do not support these functions since they only deal with specific instances of control sequences.

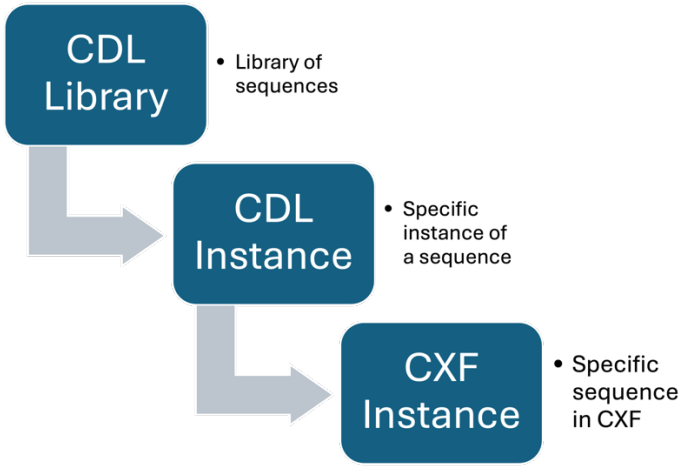


Figure 8-2 Control Sequence Libraries and Instances

8.1.3 Conformance Classes

General: Conformance classes identify groupings of functionalities. During the development and subsequent testing and validation of an application or tool, developers must determine which combination of classes to support.

Conformance Class: Conformance classes are designated as "1" and "2." Class 1 includes the functionality required for the application of a control sequence in tools intended to edit, test, and apply sequences into controllers. The functions do not include those specifically needed for support of libraries and simulation, such as vectors and matrixes. Class 2 includes all the functionality included in Class 1 plus those additional functionalities required for libraries and simulation.

Data Type Support: The conformance class also includes a second factor which is used to designate support for either "strict data types" (i.e., only Real or Integer) or "flexible data types" which allows grouping Integer and Real data type as "Analog" type. The designation "A" is used for strict data type, and the designation "B" is used for flexible data type.

Examples:

- Tools for Sequence Design – Conformance Class 2A: A tool that supports Modelica, which is being used to develop and support libraries of sequences would need to comply with Class 2 and would have to support Data Type A
- Tool for Applying Sequences – Conformance Class 1A or 1B: This could be a tool provided by a control provider which is used to import, export, edit, or define a

sequence. This tool also may have functionality to compile and download sequences into controllers. Since this tool is primarily dealing with instance and not libraries, it suffices to be type 1. If the target platform used strict data types, it would be type A, and if it uses Analog type, it would be type B. Note that these tools can also conform to other classes such as 2A or 2B.

8.2 – Associated Work

This standard is connected to a series of tools and other efforts being developed as open-source projects by the US Department of Energy. These tools are intended to comply with this standard and made available for use by industry. See the Appendix B for details. These include:

- Library of sequences in CDL that can be used or extended and modified by users.
- Tools to assist HVAC system designers in selecting a sequence from the library.
- Tools for using CDL as part of energy models.
- Tools to verify that an installed control system produces the same control response as the CDL specification.
- Tools to assist control suppliers in translating CXF to their native formats.

8.2.1 Sample CDL sequences

There is a library of sample sequences in CDL available from LBL. These include many of the sequences from ASHRAE Guideline 36 as well as other content. See https://simulationresearch.lbl.gov/modelica/releases/v12.1.0/help/Buildings_Controls_OBC_ASHRAE.html

8.2.2 Translating from CDL to CXF
Evaluation of Assignment of Values to Parameters

The assignments of values to parameters can optionally be evaluated by a CDL translator. While such an evaluation is not preferred, it is allowed in CDL to accommodate the situation that most building control product lines, in contrast to modeling tools such as Modelica, Simulink, or LabVIEW, do not support the propagation of parameters, nor do they support the use of expressions in parameter assignments.

Consider the statement
parameter Real pRel(unit="Pa") = 50 "Pressure difference across damper";

```
CDL.Continuous.Sources.Constant con(
  k = pRel) "Block producing constant output";
CDL.Logical.Hysteresis hys(
  uLow  = pRel-25,
  uHigh = pRel+25) "Hysteresis for fan control";
```

Some building control product lines will need to evaluate this at translation because they cannot propagate parameters and/or cannot evaluate expressions.
To lower the barrier for the development of a CDL translator to a control product line, the modelica-json translator has two flags. One flag, called evaluatePropagatedParameters, will cause the translator to evaluate the propagated parameter, leading to a *CDL-JSON* declaration that is equivalent to the declaration

```
CDL.Continuous.Sources.Constant con(
  k(unit="Pa") = 50) "Block producing constant output";
CDL.Logical.Hysteresis hys(
  uLow  = 50-25,
  uHigh = 50+25) "Hysteresis for fan control";
```

Note

1. The parameter Real pRel(unit="Pa") = 50 has been removed as it is no longer used anywhere.
2. The parameter con.k now has the unit attribute set as this information would otherwise be lost.
3. The parameter hys.uLow has the unit *not* set because the assignment involves an expression. As expressions can be used to convert a value to a different unit, the unit will not be propagated if the assignment involves an expression.

Another flag called evaluateExpressions will cause all mathematical expressions to be evaluated, leading to a *CDL-JSON* declaration that is equivalent to the CDL declaration

```
parameter Real pRel(unit="Pa") = 50 "Pressure difference across damper";

CDL.Continuous.Sources.Constant con(
  k = pRel) "Block producing constant output";
CDL.Logical.Hysteresis hys(
  uLow  = 25,
  uHigh = 75) "Hysteresis for fan control";
```

If both evaluatePropagatedParameters and evaluateExpressions are set, the result would be equivalent of the declaration

```
CDL.Continuous.Sources.Constant con(
  k(unit="Pa") = 50) "Block producing constant output";
CDL.Logical.Hysteresis hys(
  uLow  = 25,
  uHigh = 75) "Hysteresis for fan control";
```

Clearly, use of these flags is not preferred, but they have been introduced to accomodate the capabilities that are present in most of today's building control product lines.
[Note
A commonly used construct in control sequences is to declare a parameter and then use the parameter once to assign the value of a block in these sequences. In CDL, this construct looks like

```
parameter Real pRel(unit="Pa") = 50 "Pressure difference across damper";
CDL.Continuous.Sources.Constant con(k = pRel) "Block producing constant output";
```

Note that the English language sequence description would typically refer to the parameter pRel. If this is evaluated during translation due to the evaluatePropagatedParameters flag, then pRel would be removed as it is no longer used. Hence, such a translation should then rename the block con to pRel, e.g., it should produce a sequence that is equivalent to the CDL declaration

CDL.Continuous.Sources.Constant pRel(k = 50) "Block producing constant output";
In this way, references in the English language sequence to pRel are still valid.
]

8.2.3 Digital Documentation of CDL

The definition of CDL found in this standard is also available in a digital code repository. See https://simulationresearch.lbl.gov/modelica/releases/v12.1.0/help/Buildings_Controls_OBC_CDL.html for details.