

# **WRESL+ Language Reference**

**Draft Documentation**

**April 2013**

# Contents

WRESL Language Overview .....	5
WRESL+ Updates .....	5
Quick Start .....	6
Notation Conventions .....	7
Source Form.....	7
Variable Names and Labels.....	7
Comments.....	7
Mathematical and Logical Operators.....	8
Expressions .....	8
Value Expression .....	8
Assignment Expression .....	9
Constraint Expression .....	9
Comparison Expression .....	9
Conditional Expression .....	9
Preprocessed Expressions .....	10
Future Array Expressions .....	11
Sequence Statement .....	11
Model Statements .....	11
Include Statement .....	12
Timeseries Input Declaration .....	12
Decision Variable Declarations.....	13
Continuous Decision Variable .....	13
Bound Specification .....	13
Binary Decision Variable .....	14
Integer Decision Variable.....	14
State Variable Declarations and Assignments.....	14
Simple Value State Variable .....	15
Sum Value State Variable .....	15

Lookup Table Value State Variable .....	15
Conditional Value State Variable .....	16
Alias Variable Declaration and Assignment.....	16
Goal Statement.....	17
Concise Goal Statement .....	17
Simple Goal Statement .....	17
Conditional Goal Statement .....	18
Objective Function Statement.....	19
Standard Objective Function Statement .....	19
Common Weight Objective Function Statement.....	20
Preprocessed Statements.....	20
Initial Statement.....	20
If Statement .....	21
Future Array Statements .....	21
Future Array Decision Variable Declarations .....	22
Future Array Continuous Decision Variable.....	22
Future Array Binary Decision Variable .....	22
Future Array Integer Decision Variable.....	22
Future Array State Variable Declarations and Assignments .....	23
Future Array Simple Value State Variable.....	23
Future Array Sum Value State Variable.....	23
Future Array Lookup Table Value State Variable.....	24
Future Array Conditional Value State Variable.....	24
Future Array Alias Variable Declarations and Assignments .....	24
Future Array Goal Statements.....	25
Future Array Concise Goal Statement .....	25
Future Array Simple Goal Statement.....	25
Future Array Conditional Goal Statement.....	26
Future Array Objective Function Statement .....	27
Network Statement.....	27
Include Common Items in Multiple Sequences.....	27

Previous and Future Time Step Values ..... 28

Access Variables in Prior Sequence..... 29

Intrinsic Definitions and Functions ..... 29

    Intrinsic Functions ..... 30

    Intrinsic Definitions ..... 31

        daysin ..... 31

        month ..... 31

        wateryear ..... 31

        <MMM> ..... 31

        prev<MMM> ..... 31

## WRESL Language Overview

The Water Resources Simulation Language (WRESL) was designed to serve as flexible language interface to the linear programming (LP) solver. Specific operational rules may be specified in the WRESL language, which in turns formulates the rules into constraints and objective function terms in a form acceptable by the LP solver. Data, in the form of timeseries inputs, prior decision variable values, and relational table values, is all available to the user when formulating the operational rules.

When writing statements in the WRESL language it is important to recognize the difference between state variables and decision variables for the time period of interest. State variables (svar) have known values for the upcoming period and can be thought of as the information available to planner/operator prior to any system operation. Decision variables (dvar) are unknown and represent the decisions to be made for the upcoming period.

WRESL statements are assembled into text files with the *wres/* extension. The main WRESL file must contain the model and sequence statements that describe the constraints to be included in a particular simulation and the order in which models should be solved. Each model contains several include statements which identify the location of constraints to be considered. Operational constraints can be assembled in multiple files and placed in directories aptly named for better organization. Several WRESL language constructs are available to the user for developing constraints or rules for the water resource system and are discussed in detail below.

## WRESL+ Updates

The WRESL+ parser has replaced the old WRESL parser. The WRESL+ parser adds new functionalities and is compatible with legacy WRESL files except that the global include feature is replaced by a new approach to include the same files in multiple sequences (see [Include Common Items in Multiple Sequences](#)).

New additions of WRESL+ include:

1. Future array syntax for multi-period optimization
2. If statement for preprocessing variables and files inclusion
3. Network statement for system grids and automatic continuity generation

## Quick Start

A simple study may be composed of the following statements:

- Sequence Statement
- Model Statement
- Timeseries Input Declaration
- State Variable Declarations and Assignments (keyword is svar)
- Decision Variable Declarations (keyword is dvar)
- Goal Statement
- Objective Function Statement
- Include Statement

A simple study example is shown below.

```
sequence CYCLE1 { model First order 1 }
sequence CYCLE2 { model Second order 2 }

model First {
  timeseries I01 { kind 'FLOW-INFLOW' units 'TAF' }
  dvar X1 { std kind 'FLOW-CHANNEL' units 'CFS' }
  dvar X2 { std kind 'FLOW-CHANNEL' units 'CFS' }
  svar C { value I02*5 }
  goal Test { X1 + X2 < C }
  include 'allocation\test1.wresl'
  objective XGroup {
    [ X1, 10 ]
    [ X2, 20 ]
  }
}

model Second {
  timeseries I02 { kind 'FLOW-INFLOW' units 'TAF' }
  dvar Y1 { std kind 'FLOW-CHANNEL' units 'CFS' }
  dvar Y2 { std kind 'FLOW-CHANNEL' units 'CFS' }
  svar X1_Upstream { value X1[Upstream] }
  goal Test { Y1 + Y2 < I02 + X1_Upstream }
  include 'allocation\test2.wresl'
  objective YGroup {
    [ Y1, 10 ]
    [ Y2, 20 ]
  }
}
```

## Notation Conventions

This WRESL+ reference uses the following conventions:

1. **bold-faced** words starting with lowercase represent keywords and must be typed exactly as shown.
2. **Bold-faced** words starting with uppercase represent the rule names of statements, declarations, or expressions.
3. *Italic* words denote the identifiers, tags, or numbers of your own creation.
4. A group formed by items are enclosed within the angle brackets “{ }”
5. If one and only one of options is required, those options are shown separated by a vertical bar symbol “|”
6. If an item can be matched for zero or more times, it is followed by a star “\*”
7. If an item can be repeated at least once, it is followed by a plus sign “+”
8. If an item is optional, it’s followed by a question mark “?”
9. Braces “{ }” provide a visual grouping of statements. You may position braces and indenting according to your preference, and include optional line breaks and blank lines. You may also omit line breaks between keywords. The examples in this document show preferred styles of placement.
10. Example codes are provided inside textboxes.

## Source Form

The WRESL+ language is somewhat free-format. You may insert blank lines and comments at any point. Most statements may be on multiple lines without a continuation character. Semicolons are not allowed at the end of statements. The WRESL+ file is read sequentially, in that the statements read and process in the order that they appear in the file. Note that variables must be declared before they can be used in other statements.

## Variable Names and Labels

The WRESL+ variable name and label character set consist of letters A-Z, digits 0-9, and the “\_” character. Decision and state variable names may be up to 16 characters long. Labels and other tags may have 32 characters. Mixed case is ignored.

## Comments

Comments may be enclosed in `/*` and `*/` or introduced on a single line by `!`.

```

/* This is a
comment */

! This is another comment

```

## Mathematical and Logical Operators

Mathematical operators in the WRESL language are similar to those in other programming languages. A summary of the operators are shown below.

Operator	Description	Operator Type
+	addition	mathematical
-	subtraction	mathematical
*	multiplication	mathematical
/	division	mathematical
<	less than or equal	mathematical
>	greater than or equal	mathematical
=	equal	mathematical
<=	less than or equal	logical
>=	greater than or equal	logical
==	exactly equal	logical
.ne.	not equal	logical
.and.	logical conjunction	logical
.or.	logical inclusive disjunction	logical
.not.	logical negation	logical

In addition to the operators shown above, several intrinsic definitions and functions are available for use in WRESL statements. See [Intrinsic Definitions and Functions](#).

## Expressions

Expressions are formed by numbers, variables, functions, and operators. Value expressions are used to express value; assignment expressions are used to set the value for the variables; constraint expressions are used to set LP constraints in Goal statements; comparison expressions and conditional expressions are used to set conditions.

### Value Expression

A value expression is a single item consists of number, variable, and function; or a group of these connected by mathematical operators. For example,

5.0 ! value expression can be a single variable or number

$X + Y - Z * \max(A, B, C) / 9.0$

## Assignment Expression

An assignment expression is formed by single variable at the left hand side followed by equal sign and then value expression.

*Variable* = **Value Expression**

$X = Y + Z * \max(A, B, C)$

## Constraint Expression

A constraint expression is used to specify LP constraint inside Goal statement. It's formed by two value expressions separated by  $\langle = | > | >= | < | <= \rangle$ . Note that the double equal sign "==" is not valid in constraint expressions.

**Value Expression**  $\langle = | > | >= | < | <= \rangle$  **Value Expression**

$X = Y + Z * \max(A, B, C)$

$X + Y = Z * \max(A, B, C)$

$X + Y > Z * \max(A, B, C)$

## Comparison Expression

A comparison expression is the building block for conditional expression. It's formed by two value expressions connected by  $\langle == | > | >= | < | <= \rangle$ . Note that the single equal sign "=" is not valid in comparison expressions.

**Value Expression**  $\langle == | > | >= | < | <= \rangle$  **Value Expression**

$X + Y == Z * \max(A, B, C)$

## Conditional Expression

A conditional expression is a **Comparison Expression**, or a group of **Comparison Expression** separated by  $\langle .and. | .or. | .not. \rangle$ . Parentheses can be used to specify order

of operations. The main restriction on conditional expressions is that they may not contain any current-period decision variables.

**ComparisonExpression** ( ( **.and.** | **.or.** | **.not.** ) **ComparisonExpression** )\*

```
X + Y == Z * max(A, B, C)
X + Y >= max(A, B, C) .and. ( month > oct .or. Z < 5.0 )
```

In lieu of a conditional expression, you may specify the keyword **always**. Such a condition will always evaluate as true, and thus will always be selected unless a prior case was evaluated as true.

```
svar NMTTest {
  case February {
    condition month == feb
    value S10(-12) + sumI10_part }
  case Others {
    condition always
    value S10(prevfeb) + sumI10_part }
}
```

## Preprocessed Expressions

The value expressions in the **Initial Statement** and the conditional expressions in the **If Statement** cannot contain any time-varying or run-time evaluated variables. They are processed only once during the parsing stage and stay constant afterwards.

```
initial {
  svar A { value 10. }
  svar B { value A*5 }
  svar C { value B+6.0 }
}
```

```
if A+B>15. {
  include 'swp_dellologic\allocation\co_extfcn.wresl'
  include 'Delta\DeltaExtFuncs_7inpANN.wresl'
}
```

## Future Array Expressions

The symbol “**\$m**” can be used in the **Future Array Statements** to simplify the code. It means iteration from current timestep to the maximum future timestep.

```
S_trnty_3($m) < S_trntyLevel3adj($m) - S_trntyLevel2($m)
```

## Sequence Statement

Sequence statements define the order and optional condition in which to simulate a study consisting of multiple models. For example, it may be desirable for the modeler to operate a portion of the water resource system independent of the remaining system. In this case, the system would be divided into two models: the first includes the portion to be operated independently and the second includes the remaining part of the system. The first model is given the simulation order 1 and the second model is given the simulation order 2. The results of simulating the first model can be used in the second model. Each model must be given a unique simulation order.

```
sequence SequenceName {  
    model ModelName  
    ConditionalExpression?  
    order orderNumber  
    < timestep 1mon | 1day >?  
}
```

The conditional expression is evaluated before the sequence starts; therefore the variables in this expression must have known values. The default time step in a sequence is one month, but can be changed to be one day.

```
sequence CYCLE1 { model Upstream order 1 }  
sequence CYCLE2 { model Base order 2 timestep 1mon }  
sequence CYCLE3 { model Test order 3 timestep 1day }  
sequence CYCLE4 { model Test2 condition MON==jun order 4 }
```

## Model Statements

The model construct defines which statements will be part of a particular model formulation. Multiple models may be specified within a study given that they have a unique simulation order. All goals, objective functions, include statements, and run-time evaluated variables must be located within the model construct.

```

model modelName {
    IncludeStatement?
    TimeseriesInputDeclaration?
    StateVariableDeclarations?
    DecisionVariableDeclarations?
    AliasVariableDeclaration?
    GoalStatement?
    ObjectiveFunctionStatement?
}

```

```

model Upstream {
    include 'Delta\DeltaExtFuncs_7inpANN.wresl'
    include 'swp_dellogic\allocation\co_extfcn.wresl'
    include 'weight-table.wresl'
}

```

## Include Statement

You may include other WRESL files by inserting an include statement.

**include** 'relative wresl file path'

```

include 'Delta\DeltaExtFuncs_7inpANN.wresl'
include 'swp_dellogic\allocation\co_extfcn.wresl'

```

## Timeseries Input Declaration

The data stored in the time-series HEC-DSS database can be referenced by using the following syntax:

```

timeseries TimeseriesName {
    kind 'KindSpec'
    units 'UnitSpec'
    < convert 'ToUnits' >?
}

```

The B-part of the DSS path name will be set to the user specified *TimeseriesName*. The optional convert field will convert between **TAF** and **CFS** only. If, for example, *ToUnits* is **CFS** then a conversion will be performed from **TAF** to **CFS**.

```
timeseries I10 { kind 'FLOW-INFLOW' units 'TAF' convert 'CFS' }
timeseries I12 { kind 'FLOW-INFLOW' units 'TAF' }
```

## Decision Variable Declarations

All variables must first be declared, and thereafter used in subsequent definitions or goals. There are three types of decision variables and each type will be described separately below.

### Continuous Decision Variable

Continuous decision variables can be declared in the model using the following syntax:

```
dvar DvarName {
    std | Bound Specification
    kind 'Kindspec'
    units 'UnitSpec'
}
```

The **std** keyword indicates that the variable will have the standard LP non-negativity bounds and 1.0e38 upper bound. The **Bound** Specification, used in lieu of the keyword **std**, refers to explicit setting of the variable's lower and upper bounds (numeric constants only).

### Bound Specification

Bound specification is used to explicitly set the lower and upper bounds of the decision variable. Only numeric constants are allowed for the *UpperBound* and *LowerBound*.

```
upper UpperBound | unbounded
lower LowerBound | unbounded
```

where **unbounded** indicates  $-1.0e38$  or  $1.0e38$ .

```
dvar C_Delta_SWP {std kind 'FLOW-CHANNEL' units 'CFS' }
dvar QPD {lower -100. upper unbounded kind 'FLOW' units 'CFS'}
dvar COSZ {lower unbounded upper 9000. kind 'FLOW' units 'CFS' }
```

## Binary Decision Variable

Binary decision variables may be declared in the model by specifying **binary** as the first keyword within the braces. A binary variable can only have a value of either 0 or 1. The use of binary variables may increase computation time.

```
dvar DvarName {  
    binary  
    kind 'KindSpec'  
    units 'UnitSpec'  
}
```

```
dvar B { binary kind 'BINARY' units 'NA' }
```

## Integer Decision Variable

Integer decision variables may be declared in the model by specifying **integer** as the first keyword within the braces. The standard bounds for an integer variable are the same as for a binary variable (0 or 1). The use of integer variables with non-standard bounds is discouraged because they may greatly increase computation time.

```
dvar DvarName {  
    integer  
    std | BoundSpecification  
    kind 'KindSpec'  
    units 'UnitSpec'  
}
```

```
dvar Integer1 { integer std kind 'BINARY' units 'NA' }  
dvar Integer2 { integer lower 0 upper 3 kind 'INTEGER' units 'NA' }
```

## State Variable Declarations and Assignments

State variables are used to hold information that may be used in expressions within subsequent state variable assignments, goal statements, objective function statements, alias statements, or future array statements. There are several types of state variables and each type will be described separately below.

## Simple Value State Variable

The simple value assignment uses the **Value Expression** to set the value for state variables.

```
svar SvarName {  
    value Value Expression  
}
```

```
svar X { value 9.0 }  
svar Y { value max(X, 5.0) }
```

## Sum Value State Variable

The sum value assignment specifies the state variable value to be the sum of an expression using an iterator “i”. *iBegin* is the beginning index, *iEnd* is the ending index, and *iStep* is the step size. This is especially useful for summing values of timeseries variables using this syntax to represent the time offset:

```
svar SvarName {  
    sum( i = iBegin, iEnd <, iStep >? ) Value Expression  
}
```

```
svar Z { sum(i=1, 5, 1) S03(-i) + I10(-i) }  
svar OroDivEst { sum(i=0, sep-month, 1) D_PWR(i) }
```

## Lookup Table Value State Variable

This syntax retrieves a value from a relational database table. It’s a simplified version of the SQL language. You can specify either or both of the **where** clause or the **given/use** clauses.

```
svar SvarName {  
    select Result from Tablename  
    < given Assignment Expression use < linear | max | min >? >  
    < where Assignment Expression+ >?  
}
```

```
svar D_M {select DCU_M from DCU_MAC where MON=jan}
```

## Conditional Value State Variable

State variable values may be assigned based upon user specified conditions.

```
svar svarName {  
    case ConditionLabel1 {  
        condition Conditional Expression  
        SimpleValue, SumValue, or LookupTableValue Assignment }  
  
    case ConditionLabel2 {  
        condition Conditional Expression  
        SimpleValue, SumValue, or LookupTableValue Assignment }  
  
    .....  
  
    case DefaultConditionLabel {  
        condition always  
        SimpleValue, SumValue, or LookupTableValue Assignment }  
}
```

The *ConditionLabel* is an alphanumeric tag that will be used by the program to label which condition is being used. The **Conditional Expressions** are evaluated in the order specified until one evaluates to true. The final condition must be specified as **always** for a default case.

```
svar NMTest {  
    case February {  
        condition month == feb  
        value S10(-12) + sumI10_part }  
    case Others {  
        condition always  
        value S10(prevfeb) + sumI10_part }  
}
```

## Alias Variable Declaration and Assignment

Alias variable can be defined as a linear combination of previously defined decision variables. It can be used as a variable in **Goal Statement** and its value will be available in the output HEC-DSS file. These help a great deal with brevity and are useful for understanding the model's output. If an alias variable is used in the goal statements, it will be declared as unbounded upper and lower.

```
alias aliasName {  
    Value Expression
```

```

    kind 'Kindspec'
    units 'UnitSpec'
}

```

```

! EstSodExp is a previously defined decision variable
alias EstSod { EstSodExp kind 'ESTIMATE' units 'CFS' }

! Est_rel is a variable with known value
alias Estlim3 { max(300.,Est_rel) kind 'DEBUG' units 'CFS' }

```

## Goal Statement

Goal statements are used to specify system operating constraints. They can be specified as “hard” constraints (LP constraint must be satisfied) or as “soft” constraints (LP constraint may be violated, subject to some penalty). A right-hand-side (RHS) and a left-hand-side (LHS) can be used to specify the constraints.

## Concise Goal Statement

The concise form of the goal statement uses the **Constraint Expression**. This is a preferred method for specifying constraints without conditionals or penalties.

```

goal GoalLabel {
    Constraint Expression
}

```

```

goal Test { X > Y }

```

## Simple Goal Statement

If there are no associated conditionals, then you can use this simple form. If the **lhs>rhs** or **lhs<rhs** statement is omitted, or the keyword **constrain** is used instead of **penalty**, the specified constraint will be a hard constraint. There will be a bound applied, and the solver will not be allowed to violate it. Note that “<” means “<=” and “>” means “>=” in LP lingo. Penalty can be a numeric constant or an expression consisting of variables with known values.

```

goal GoalLabel {
    lhs Value Expression
    rhs Value Expression
    < lhs>rhs < constrain | < penalty Value Expression > >?
}

```

```

    < lhs<rhs < constrain | < penalty Value Expression >>? > }
}

```

```

goal NMTTest {
    lhs X+Y
    rhs Z
    lhs>rhs constrain
    lhs<rhs penalty 0
}

```

## Conditional Goal Statement

The constraints can be specified based on the **Conditional Expressions**. They will be evaluated in the order they are presented until the program reaches one that evaluates to true.

```

goal GoalLabel {
    lhs Value Expression
    case CaseLabel1 {
        condition Conditional Expression
        rhs Value Expression
        < lhs>rhs < constrain | < penalty Value Expression >>?
        < lhs<rhs < constrain | < penalty Value Expression >>? }
    case CaseLabel 2 {
        condition Conditional Expression
        rhs Value Expression
        < lhs>rhs < constrain | < penalty Value Expression >>?
        < lhs<rhs < constrain | < penalty Value Expression >>? }
    ...
    case DefaultCaseLabel {
        condition always
        rhs Value Expression
        < lhs>rhs < constrain | < penalty Value Expression >>?
        < lhs<rhs < constrain | < penalty Value Expression >>? }
}

```

```

goal TestAction1 {
  lhs D_J1 + D_B1
  case April {
    condition month == apr
    rhs      CNW + X
    lhs<rhs  penalty 0 }

  case Others {
    condition always
    rhs      3000.*16.0/daysin + Z
    lhs<rhs  penalty 0
    lhs>rhs  constrain }
}

```

## Objective Function Statement

The LP solver will try to find one feasible solution that maximizes the objective function composed of decision variables and their coefficients. The uniqueness of decision variable solutions is not guaranteed unless the maximum objective value only occurs at a single point on the feasible region. The decision variable solution returned by the LP solver is probably only one of infinitely many solutions that give the same maximum objective value.

## Standard Objective Function Statement

In the standard objective function statement, the first term in the brackets is the decision variable name, and the second term is the coefficient (or weight) of this decision variable.

```

objective ObjectiveGroupLabel {
  [ VariableName1, Value Expression ]
  [ VariableName2, Value Expression ]
  ...
  [ VariableNameN, Value Expression ]
}

```

```

objective XGroup {
    [ X1, 10 ]
    [ X2, 10 ]
    [ X3, 10 ]
}

objective YGroup {
    [ Y1, 10*Z ]
    [ Y2, 20*Z ]
}

```

## Common Weight Objective Function Statement

If a group of decision variables have the same weight in the objective function, the common weight statement can be used to avoid the repetitions.

```

objective ObjectiveGroupLabel {
    weight Value Expression
    variable variable1 variable2 ... variableN
}

```

```

objective XGroup {
    weight 10
    variable X1 X2 X3
}

```

## Preprocessed Statements

The **Initial Statement** and **If Statement** are processed only once during the parsing stage and will not be revisited during the study runtime. In the parsing stage, the variables inside the Initial Statement are evaluated first so the If Statements knows which files to include.

### Initial Statement

The Initial statement currently allows only **Simple Value State Variable** and **Lookup Table Value State Variable**. These variables are processed only once and will not be re-evaluated after the parsing stage. They can be referenced in the **If Statement** to preprocess file inclusions.

```
initial {
    Simple Value State Variable | Lookup Table Value State Variable
}
```

```
initial {
    svar A { 100 }
    svar B { value A - 5 }
    svar C { select OptionK from Options where IndexB = 2 }
}
```

## If Statement

If statement can be used to preprocess file inclusions. The dependent variables in the conditional expressions must be defined in the **Initial Statement**.

```
if PreprocessedConditionalExpression {
    Include Statement+ }
< elseif PreprocessedConditionalExpression {
    Include Statement+ }>?
< else PreprocessedConditionalExpression {
    Include Statement+ }>?
```

```
if A + B > C {
    include 'swp_delloic\allocation\co_extfcn.wresl'
}
```

```
if Scenario == 1 {
    include 'allocation\case1.wresl'
} elseif Scenario == 2 {
    include 'allocation\case2.wresl'
} else {
    include 'allocation\case3.wresl'
}
```

## Future Array Statements

Future array statements can simplify coding for multi-period optimizations by using arrays and an iterator to define variables and constraints efficiently

## Future Array Decision Variable Declarations

This syntax can declare and assign an array of decision variables with minimum repetition. The Future Array Decision Variable Declarations are similar to the standard **Decision Variable Declarations** except the *FutureArraySize* is in parenthesis. *FutureArraySize* allows only either numeric constant or a state variable that is assigned to a numeric constant. For example, *FutureArraySize* of 6 will generate 6 future variables and one current variable. There are three types of decision variables (shown below).

### Future Array Continuous Decision Variable

```
dvar(FutureArraySize) DvarName {  
    std | Bound Specification  
    kind 'Kindspec'  
    units 'UnitSpec'  
}
```

```
dvar(11) C_Delta_SWP {std kind 'FLOW-CHANNEL' units 'CFS' }  
dvar(23) QPD {lower -100. upper unbounded kind 'FLOW' units 'CFS' }
```

```
svar FAMmonths { value 11 }  
dvar(FAMmonths) C_Delta_SWP {std kind 'FLOW-CHANNEL' units 'CFS' }
```

### Future Array Binary Decision Variable

```
dvar(FutureArraySize) DvarName {  
    binary  
    kind 'Kindspec'  
    units 'UnitSpec'  
}
```

```
dvar(35) B { binary std kind 'BINARY' units 'NA' }
```

### Future Array Integer Decision Variable

```
dvar(FutureArraySize) DvarName {  
    integer  
    std | Bound Specification  
    kind 'Kindspec'
```

```

    units 'UnitSpec'
}

```

```

dvar(35) Integer1 { integer std kind 'BINARY' units 'NA' }
dvar(47) Integer2 { integer lower 0 upper 3 kind 'INT' units 'NA' }

```

## Future Array State Variable Declarations and Assignments

This syntax can declare and assign an array of state variables with minimum repetition. The Future Array State Variable Declarations are similar to the standard [State Variable Declarations and Assignments](#) except for the *FutureArraySize* in the parenthesis and the special symbol **\$m** in the expression. *FutureArraySize* only allows either a numeric constant or a state variable that is assigned to a numeric constant. For example, *FutureArraySize* of 6 will generate 6 future variables and one current variable. The special symbol **\$m** is used to iterate the expression from 0 to *FutureArraySize*. There are four types of state variables (shown below).

### Future Array Simple Value State Variable

```

svar(FutureArraySize) SvarName {
    value ValueExpression
}

```

```

svar FAMmonths { value 23 }
svar(FAMmonths) wy { value int(( month + $m-1 )/12 ) + wateryear }

```

### Future Array Sum Value State Variable

```

svar(FutureArraySize) SvarName {
    sum(i = iBegin, iEnd (, iStep)? ) ValueExpression
}

```

```

svar(23) Z { sum(i=-month+X($m)*12, 8-month+X($m)*12, 1)
            D27A(i)*cfs_taf(i)+
            D85B(i)*cfs_taf(i)+
            D88C(i)*cfs_taf(i) }

```

## Future Array Lookup Table Value State Variable

```
svar(FutureArraySize) SvarName {  
    select Result from Tablename  
    < given AssignmentExpression use < linear | max | min > >?  
    < where AssignmentExpression+ >?  
}
```

```
svar(11) D_M {select DCU_M from DCU_MAC where MON=mv($m)}
```

## Future Array Conditional Value State Variable

```
svar(FutureArraySize) svarName {  
    case ConditionLabel1 {  
        condition ConditionalExpression  
        value assignment }  
  
    case ConditionLabel2 {  
        condition ConditionalExpression  
        value assignment }  
  
    .....  
  
    case DefaultConditionLabel {  
        condition always  
        value assignment }  
}
```

```
svar(FAMMonths) mv{  
    case September {  
        condition (month+$m)/12.0==int((month+$m)/12.0)  
        value 12  
    }  
    case OctToAug {  
        condition always  
        value (month+$m)-int((month+$m)/12.0)*12  
    }  
}
```

## Future Array Alias Variable Declarations and Assignments

```
alias(FutureArraySize) AliasName {  
    ValueExpression
```

```

    kind 'Kindspec'
    units 'UnitSpec'
}

```

```
dvar(2) Rio_V {std kind 'FLOW' units 'CFS' }
```

```
alias(2) Mif_R { Rio_V($m) kind 'FLOW' units 'CFS' }
```

The above statements will generate total 3 dvars and 3 alias:

```
dvar Rio_V {std kind 'FLOW' units 'CFS' }
```

```
dvar Rio_V__future__1 {std kind 'FLOW' units 'CFS' }
```

```
dvar Rio_V__future__2 {std kind 'FLOW' units 'CFS' }
```

```
alias Mif_R { Rio_V 'FLOW' units 'CFS' }
```

```
alias Mif_R__future__1 { Rio_V__future__1 'FLOW' units 'CFS' }
```

```
alias Mif_R__future__2 { Rio_V__future__2 'FLOW' units 'CFS' }
```

## Future Array Goal Statements

This syntax can specify an array of constraints with minimum coding. The Future Array Goal Statements are similar to the standard **Error! Reference source not found.** except for the *FutureArraySize* in the parenthesis and the special symbol **\$m** in the constraint expression. *FutureArraySize* only allows either a numeric constant or a state variable that is assigned to a numeric constant. For example, *FutureArraySize* of 6 will generate 6 future goals and one current goal. The special symbol **\$m** is used to iterate the expression from 0 to *FutureArraySize*.

## Future Array Concise Goal Statement

```
goal(FutureArraySize) GoalLabel {
    ComparisonExpression
}
```

```
goal(23) set_N_SWP { D929($m) > max(700., 0.125*AD_R($m)) }
```

## Future Array Simple Goal Statement

```
goal(FutureArraySize) GoalLabel {
    lhs ValueExpression
}
```

### rhs ValueExpression

< lhs>rhs < constrain | < penalty ValueExpression > > > ?  
< lhs<rhs < constrain | < penalty ValueExpression > > > ? }

}

```
goal(23) set_N_SWP {  
    lhs D929($m)  
    rhs max(700., 0.125*AD_R($m))  
    lhs<rhs penalty 0  
}
```

### Future Array Conditional Goal Statement

goal(FutureArraySize) GoalLabel {

#### lhs ValueExpression

case CaseLabel1 {

condition Conditional Expression

#### rhs ValueExpression

< lhs>rhs < constrain | < penalty Value Expression > > > ?

< lhs<rhs < constrain | < penalty Value Expression > > > ? }

case CaseLabel 2 {

condition Conditional Expression

#### rhs ValueExpression

< lhs>rhs < constrain | < penalty Value Expression > > > ?

< lhs<rhs < constrain | < penalty Value Expression > > > ? }

...

...

case DefaultCaseLabel {

condition always

#### rhs ValueExpression

< lhs>rhs < constrain | < penalty Value Expression > > > ?

< lhs<rhs < constrain | < penalty Value Expression > > > ? }

}

```

goal(FAMmonths) LimitR{
  lhs C129($m)
  case Janurary {
    condition mv($m)==jan .and. int(BOON_SWP)==1
    rhs W_2_OMR_target_Jan
    lhs>rhs penalty 0 }
  case February {
    condition mv($m)==feb .and. int(BOON_SWP)==1
    rhs W_2_OMR_target_Feb
    lhs>rhs penalty 0 }
  case Default {
    condition always
    rhs W_1_OMR_target_Others
    lhs>rhs penalty 0 }
}

```

## Future Array Objective Function Statement

```

objective ObjectiveGroupLabel {
  [ VariableName1(FutureArraySize), VariableWeightExpression ]
  [ VariableName2(FutureArraySize), VariableWeightExpression ]
  ...
  [ VariableNameN(FutureArraySize), VariableWeightExpression ]
}

```

```

objective XGroup {
  [ X1(23), 10 ]
  [ X2(23), 20 ]
  [ X3(23), 30 ]
}

```

## Network Statement

A network statement can make the connections between elements easier to read. It's currently under development.

## Include Common Items in Multiple Sequences

The group statement can be used to include the same files or variables in multiple sequences. For example, common files can be organized in a group and then that group can be included later in other models. In the example codebox below, the common files

and variables are organized into the group labeled “CommonGroup”, and are included later in the two models “First” and “Third”.

```
group GroupLabel {  
    Include Statement?  
    Timeseries Input Declaration?  
    State Variable Declarations and Assignments?  
    Decision Variable Declarations?  
    Alias Variable Declaration and Assignment?  
    Goal Statement?  
    Objective Function Statement?  
}
```

```
sequence CYCLE1 { model First order 1 }  
sequence CYCLE2 { model Second order 2 }  
sequence CYCLE3 { model Third order 3 }  
group CommonGroup {  
    dvar X { std kind 'FLOW-CHANNEL' units 'CFS' }  
    dvar Y { std kind 'FLOW-CHANNEL' units 'CFS' }  
    include 'common\common1.wresl'  
    include 'common\common2.wresl'  
}  
model First {  
    include group CommonGroup  
    include 'first\abc.wresl'  
}  
model Second {  
    include 'second\cde.wresl'  
}  
model Third {  
    include group CommonGroup  
    include 'third\efg.wresl'  
}
```

## Previous and Future Time Step Values

In state variable and goal statements, you may refer to the values of decision, alias, state, and input timeseries variable up to 12 time periods prior to the current period.

Additionally, input timeseries variables may be accessed up to 12 time periods ahead of the current period. Thus, the resulting values from previous time periods are available for determining the state of the system and future input values are available for forecasting

purposes. This time period offset access is performed by adding a suffix to the variable name, of the form:

*VariableName*(**OffsetExpression**)

For example, **Storage11(-2)** refers to the variable **Storage11** two time steps ago; and **Inflow10(4)** refers to the variable **Inflow10** four time steps ahead. **Offset Expression** uses the same rules as **Value Expression** and must evaluate negative for decision and alias variables. Omitting the suffix in parentheses causes the parser to use the value from the current model time step.

```
svar A { value S11(-12) }
svar B { value I10(3) }
```

## Access Variables in Prior Sequence

Multiple models can be specified to be simulated in a particular order. The modeler decides which parts of a system should be included in which models and the order in which to simulate the models. The decision variable results of earlier sequences are accessible in the later sequences by using the variable name followed by [*modelName*].

```
sequence CYCLE1 { model Upstream order 1 }
sequence CYCLE2 { model Base order 2 }
model Upstream {
    include 'upstream\abc.wresl'
    dvar X { std kind 'FLOW' units 'CFS' }
}
model Base {
    svar Y { value X[Upstream]+100. }
    include 'base\cde.wresl'
}
```

## Intrinsic Definitions and Functions

The following definition and function names are reserved words in WRESL+ and may not be redefined. They can be used in state variable value assignments, conditional expressions, or any other expressions. However, decision variables may not be used as arguments in any of the functions because decision variables' values are unknown before the LP problem is solved.

## Intrinsic Functions

Name	Description
cfs_taf(offset)	Returns a multiplier that converts a given quantity in cubic feet per second to thousands of acre-feet per time step where the optional "offset" is relative time step position from the current time step.
taf_cfs(offset)	Returns a multiplier that converts a given quantity in thousands of acre-feet per time step to cubic feet per second where the optional "offset" is relative time step position from the current time step.
pow(a,b)	Returns the value of "a" raised to the power of "b"
abs(a)	Returns the absolute value of "a"
int(a)	Returns an integer representation of "a"
real(a)	Returns a real/float representation of "a"
exp(a)	Returns the exponential of "a" ... $e^a$
log(a)	Returns the natural logarithm of "a"
log10(a)	Returns the common logarithm of "a"
sqrt(a)	Returns the square root of "a"
max(a,b,c,...)	Returns the maximum value of a,b,c ...
min(a,b,c,...)	Returns the minimum value of a,b,c ...
mod(a,b)	Returns the remainder of "a" divided by "b"

## Intrinsic Definitions

### daysin

The keyword **daysin** returns number of days in the current time step.

```
svar xyz {value daysin} ! in January xyz equals 31
```

### month

The keyword **month** returns the current month number of the water year in California.

```
svar xyz {value month} ! in January xyz equals 4
```

### wateryear

The keyword **wateryear** returns the current four-digit water year in California.

```
svar xyz {value wateryear} ! in October, 1921 xyz equals 1922
```

### <MMM>

*MMM* is the three-letter abbreviation of the months, e.g., **oct**, **nov**, **dec**, **jan**, **feb**. It returns the month number of the water year in California.

```
svar xyz1 {value oct} ! xyz1 equals 1  
svar xyz4 {value jan} ! xyz4 equals 4
```

### prev<MMM>

The keyword **prev** is followed by the three-letter abbreviation of the month, e.g., **prevOct**, **prevNov**, **prevDec**, **prevJan**. This returns the time step position of the specified previous month relative to the current month.

```
svar xyz {value prevOct} ! in December xyz equals -2  
svar xyz {value prevJan} ! in December xyz equals -11  
svar xyz {value prevDec} ! in December xyz equals -12
```