

# **CALSIM**

## **Water Resources Simulation Model**

### **WRESL Language Reference**

### **Draft Documentation**

## Language Overview

The Water Resources Simulation Language (WRESL) was designed to serve as flexible language interface to the linear programming (LP) solver and databases. Specific operational rules may be specified in the WRESL language, which in turns formulates the rules into constraints and objective function terms in the form acceptable to the LP solver. Data, in the form of timeseries input, prior decision variable values, and relational table values, are 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. These variables have known constant values for the upcoming period and can be thought of as the information available to planner/operator prior to any system operation. Decision variables, on the other hand, are unknown and represent the decisions to be made for the upcoming period. Thinking in this fashion will greatly expedite the learning of this language.

WRESL statements are assembled into simple text files with the *wresl* 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 *wresl* files and placed in directories aptly named for better organization. Several WRESL language constructs are available to the user in developing constraints or rules for the water resource system and are discussed in detail below.

## Notational Conventions

**Bold-faced** words or characters represent key words and must be typed exactly as shown. *Italic* words denote the names of identifiers or tags of your own creation. If one of several options is required, those options are shown separated by a vertical bar ( | ) symbol; if any item is itself optional, that item appears within **[brackets]**. 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.

## Names and Labels

The WRESL 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 (like Fortran).

## Mathematical and Logical Operators

Mathematical operators in the WRESL language are not as widely used as, but are similar to, those in other programming languages. Logical operators are used to evaluate conditional expressions and follow the Fortran90 syntax. A summary of the operators are shown below:

<b>Operator</b>	<b>Description</b>	<b>Operator Type</b>
+ and -	addition and subtraction	mathematical
* and /	multiplication and 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
.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.

### **Comments**

Comments may be the C/Java format (enclosed in /\* and \*/ , or introduced on a single line by // ) or the Fortran 90 style (introduced on a single line by '!').

### **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.

### **Statements**

There are current five kinds of statements in the WRESL language with variations on each of these. The major statements are listed below followed by a description and syntax:

- SEQUENCE
- MODEL
- INCLUDE
- DEFINE
- GOAL

### ***SEQUENCE Statements***

Sequence statements define the order and optional condition in which to simulate a study consisting of multiple models. For example, it may be desirable to the modeler to operate a portion of the water resource system independent of the remaining system. In this case, the system may be divided into two models: the first including the portion to be operated independently and the second including the remaining part of the system. The first model

could then be given the simulation order 1 and the second simulation order 2. The results of simulating the first model could then be used in the modeling of the second model. Each model must be given a unique simulation order.

```
sequence identifier {  
    model modelName  
    [condition conditionalExpression]  
    order orderNumber  
}
```

### **MODEL Statement**

The model construct defines which statements will be part of a particular model formulation. Again, multiple models may be specified within a study given that they have a unique simulation order. All define, goal, and include statements must be located within the model construct.

```
model modelName {  
    [INCLUDE Statements]  
    [DEFINE Statements]  
    [GOAL Statements]  
}
```

### **INCLUDE Statement**

You may include other WRESL files by inserting and include statement within other WRESL files. The include statement is similar to Fortran include statements and may be inserted at the top of a file or between other statements (although, this is not recommended).

```
include 'filename'
```

### **DEFINE Statements**

The define statement is similar to variable assignments and declarations in other languages. All variable identifiers must first be declared in a define statement, and thereafter used in subsequent definitions or goals. It is rather overextended in the current version, so several variations exist. Each variation will be described separately below.

#### **Continuous Decision Variable Declarations**

This form is used to add continuous (non-integer) LP *decision variables* to the solver's formulation. Decision variables are defined using the options below.

```
define yourIdentifier {  
    std | boundSpec  
    kind 'kindspec' ! DSS C-part
```

```

    units 'unitspec' ! DSS units
}

```

The **std** indicates that the variable will have the standard LP nonnegativity bounds and 1.0e38 upper bound. Most variables used will be in this category. The *boundSpec*, used in lieu of the keyword **std**, refers to explicit setting of the variable's lower and upper bounds (numeric constants only) and follows the syntax below:

```

upper constant | unbounded
lower constant | unbounded

```

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

### Binary Integer Decision Variable Declarations

Binary integer LP variables may be included in your model by specifying **integer** as the first keyword within the braces. The standard bounds for an integer variable are between 0 and 1.

```

define yourIdentifier {
    integer
    std
    kind 'kindspec' ! DSS C-part
    units 'unitspec' ! DSS units
}

```

The use of integer variables is discouraged because they may greatly increase computation time.

### State Variable: Time-Series Input Declarations and Assignments

You may refer to data stored in the input time-series (DSS) database. These values are utilized in a fashion similar to decision variables from previous time steps. You must first define the DSS pathname parts to be used for the variable using the following form of the define statement:

```

define yourIdentifier {
    timeseries ['b-part']
    kind 'kindspec' ! DSS pathname C-part
    [units 'unitspec' ] ! DSS units.
    [convert 'toUnits' ] ! Units to convert to
}

```

The B-part of the DSS path name will be set to the name specified as *yourIdentifier*. The optional b-part specification, after the *timeseries* keyword, is used to override the default

b-part name. The optional convert field will convert between TAF and CFS only. If 'toUnits' is 'CFS' then a conversion will be performed from TAF to CFS.

### External Function Declarations

External functions may be incorporated into the model through the use of an external function declaration. Declarations must occur before the function is used in a value statement. LF90 and C++ functions are permitted. For LF90, the code must be compiled and the object code located in the external directory. For C++, the code must be compiled into a dynamic-link-library and a LF90 function written to import the DLL. These files must then be placed in the external directory together with the LIB file. The names of the object and LIB files must be the same as the function name used in the WRESL statement. External functions must be specified as the lf90 or dll variety. lf90 is the default type.

```
define functionName {  
    external  
    [lf90 ] | [dll ] ! type of function  
}
```

### State Variable: Constant Value Assignments

Constant value assignments are used to hold intermediate state variable information that may be used in subsequent **goal** or conditional statements. All variable value definitions are held constants during the LP solution process as state variables.

```
define identifier {  
    value expression  
}
```

### State Variable: Table Lookup Value Assignments

This form retrieves a value from a relational database table. The facility is quite powerful and flexible.

```
define identifier {  
    select result  
    from tablename  
    [ given field=valueExpr  
    use linear | max | min ]  
    [ where field=valueExpr [,field=valueExpr] ... ]  
}
```

Specify the **use** clause when and only when you specify the **given** clause. You can specify either or both of the **where** clause or the **given/use** clauses. This statement form will be translated into a function call that retrieves data from a relational database table.

### State Variable: Sum Value Assignment

The sum value assignment allows you to specify the definition to be equal to the sum of an expression while using an index. *i* is the counter, *ibeg* is the initial value, *iend* is the stop criteria, and *istep* is the step size for this DO-LOOP type structure. This is especially powerful for summing values of timeseries variables using the index to represent the time offset:

```
define identifier {  
    sum(i=ibeg,iend[,istep]) expression  
}
```

### Value Assignment Based On Conditions

State variable values may be assigned based upon whether one or several conditional statements are true. An example of a conditional value definition is below. (Remember, the vertical bar “|” means “or,” that is, you may type either the items to the left or to the right of the bar, not the bar itself and not both).

```
define userIdentifier {  
    case ConditionIdentifier1 {  
        condition conditionalExpression  
        constant or lookup table value assignment  
    }  
    case ConditionIdentifier2 {  
        condition conditionalExpression | always  
        constant or lookup table value assignment  
    }  
    ...  
}
```

ConditionIdentifier is an alphanumeric tag that will be used by the program to label which condition was being used during that time step. Each conditionalExpression is evaluated in the order they are specified, until one evaluates as true. Subsequent statements are skipped. This is just like a Fortran if-then-elseif-endif block.

ConditionalExpression is an equation or an inequality, or a group of these separated by **.and.**, **.or.**, or **.not.** Parentheses are allowed, as are intrinsic functions. The main restriction on ConditionalExpression is that it may not contain any current-period decision variables. In lieu of a conditionalExpression, you may specify the keyword **always**. Such a condition will always evaluate true, and that case will always be selected unless a prior case evaluated true.

The conditions are evaluated in the order specified until one evaluates to true. When this happens, the userIdentifier will be symbolically equated to the expression is the value assignment. The final condition must be specified as **always** for a default case.

Note that these are not operating constraints or targets. All value assignments are merely definitions of terms that will be used later to specify constraints or targets in the goals section.

### **Alias Variable Declaration and Assignment**

You can also define a new variable as an *alias* decision variable. These help a great deal with brevity as well as being useful for understanding the model's output. You may include linear combinations of other *previously defined* decision variables here.

```
define userIdentifier {  
    alias expression  
    kind 'kindspec' ! DSS C-part  
    units 'unitspec' ! DSS units  
}
```

You may use this decision variable in Goal statements, or not if you wish. In either case, its value will be available in the output DSS database for post-execution reporting. By default, alias variables are unbounded upper and lower.

### **GOAL Statement**

Goal statements are used to specify system operating constraints and targets and are directly translated into LP constraints. Goals can be specified as having definitions that might change each time step, just like for value definitions. In addition, goals can be specified as “hard” constraints in which the LP constraint *must* be satisfied, or as “soft” constraints in which the LP constraint may be violated subject to some penalty.

These goals are expressed in the form of an equation or an inequality. In WRESL, as in other LP simulation languages, a right-hand-side (RHS) and a left-hand-side (LHS) of the constraint are specified. Optionally penalties may be assigned for violation of the constraint. Develop your expressions so that they make sense using the LHS/RHS concept: outflow = inflow; delivery <= demand; inflow - release = change in storage.

The words “LHS” and “RHS” are for convenience only and are different from the theoretical LP meaning of the terms. Decision variable terms and constant terms may both appear on either the left or right side of WRESL expressions.

The goal statement allows for a long and short form.

#### **Goal Statement: Long Form**

This is where objectives and constraints on the underlying model are specified. We are presently using an LP solver, meaning that every goal must be a linear expression of decision variables.

Here is the long format of a goal.

```
goal goalIdentifier {
```

```

lhs expression

case ConditionIdentifier1 {
    condition    conditionalExpression | always
    rhs          expression;
    [lhs>rhs      penalty penaltyExpr | never | constrain]
    [lhs<rhs      penalty penaltyExpr | never | constrain]
}
case ConditionIdentifier1 {
condition    conditionalExpression | always
rhs          expression;
[lhs>rhs      penalty penaltyExpr | never | constrain]
[lhs<rhs      penalty penaltyExpr | never | constrain]
}..
}

```

The same restrictions are in force for the *expression* terms here as for the *alias define* statement. The parser will forbid inappropriate use of decision variables.

Conditions will be evaluated in the order they are presented until the program reaches one that evaluates to true, similar to the *value define* statement. The keyword **always** is equivalent to Fortran .true.

If the **lhs>rhs** or **lhs<rhs** statement is omitted, or **never** or **constrain** is specified 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 values are generally expressed using constant values and are often zero.

There is a shorthand form, similar to the define shorthand form. If there are no associated conditionals, then you can omit the *conditionIdentifier* and the attached baggage, and simply write,

```

goal goalIdentifier {
    lhs          expression
    rhs          expression
    [lhs>rhs      penalty penaltyExpr | never | constrain ]
    [lhs<rhs      penalty penaltyExpr | never | constrain ]
}

```

### **Goal Statement: Short Form**

There is an even more concise form for expressing traditional LP constraints. Omit all tags except the goal tag and specify a comparison expression using <, >, or =. This is the preferred method for specifying constraints without associated conditions or penalties.

```
goal tagname {  
    expr1 < | = | > expr2  
}
```

### **Previous Time Step Values**

In define and goal statements, you may refer to the values of any 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 decision variable name, of the form

VariableName(offsetExpression)

For example, S11(-1) might refer to the storage in reservoir 11 one time step ago. *OffsetExpression* must of course evaluate negative for decision variables and aliases, and must be nonzero. This same form is used for timeseries input variables. Omitting the suffix in parentheses causes WRESL to use the value from the current model time step. You can specify both previous and future values (future values are positive).

### **Model Scoping Issues**

CALSIM allows the decision maker to specify multiple models 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 each higher order model (ie. simulated prior to the current model) are accessible in the current model by using the variable name followed by **[modelName]**.

WRESL also includes scope, a term used to describe whether particular statements and variables are accessible outside of the parent model. The bracketed keywords **[local]** and **[global]** are used for this purpose. If keywords **include**, **define**, or **goal** are directly followed by **[local]** then this statement will only apply to the model in which the statement exists. However, if this keyword is absent (or **[global]** of specified) the statements apply to the current model and all subsequent models.

### ***Intrinsic Definitions and Functions***

The following definition and function names are reserved words in WRESL. They are intrinsic to CALSIM and may not be redefined. Use them in any state variable value assignments, conditional expressions, or any other expression. However, decision variables may not be used as arguments in any of the functions as this would imply the variable's value is already known (which by definition it is not).

<b>Definition/Function</b>	<b>Description</b>
daysin	Number of days in current time step
month	Month number of the water year (October=1, January=3)
wateryear	Four-digit water year, runs Oct.-Sept.
MMM	Month number of water year (OCT=1,JAN=3)
prevMMM	Previous month number of water year (prevOCT=-2 in DEC)
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 timestep.
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"