Technical Documentation Version 7.0

RPL Debugging and Analysis Tools

RPL Debugging and Analysis Table of Contents

Types of RPL Debugging and Analysis

1. Types of RPL Debugging and Analysis

When building and executing RPL policy, there are three situations when debugging and analysis is required:

- When you encounter when building or validating RPL logic
- When you encounter an error when executing RPL logic
- When RPL logic produces the incorrect result or you wish to understand the RPL logic in more detail

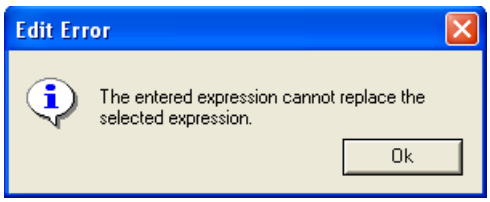
Within this section is an overview of each situation. Then, in the remainder of the document, are the tools that can be used for debugging including the RPL Debugger, Diagnostics, Run Analysis, and RPL Analysis.

1.1 Building and Validation Errors

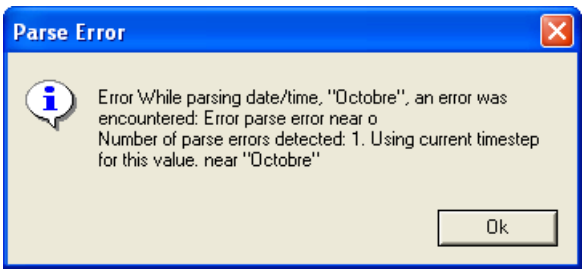
There are several errors (non-evaluation errors) which may be encountered when building or validating RPL sets. These messages do not immediately affect a model run, but will likely result in a run failure if not addressed. Messages may appear in the Diagnostics Output Window or in an error notification window which appears at the time of the error.

1.1.1 Errors when building RPL expressions

When an unspecified expression is filled in by typing text into its text field, the entry is parsed to ensure that it meets the requirements of the desired type. If the entry is not valid, a diagnostic message is generated. For example, typing **@"t"** into an unspecified **<numeric expr>** would generate the following message and revert the expression to its prior, unspecified, state:



Another kind of parsing error can occur when the correct expression data type is improperly entered. For example, typing **@"Octobre"** into an unspecified **<datetime expr>** would generate the following message and revert the expression to its prior, unspecified, state:



In both of these cases, the entered expression was incorrect. Examination of the expression and the diagnostics message should point out the flaw.

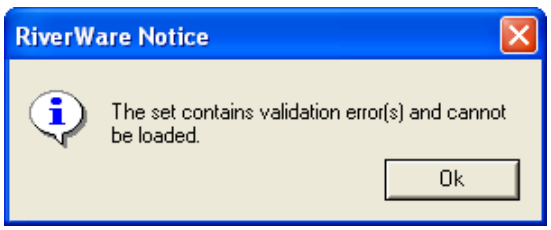
1.1.2 Errors during RPL set Validation

RPL sets are validated at several stages prior to a run. Validation ensures that a RPL set meets a minimum level of "correctness" for its intended use. There are different levels of validity required for different stages of RPL set opening, editing, and running. The validity levels are:

- **Hopeless**: A hopeless RPL set is one which cannot be read. This can occur if the RPL set file has been incorrectly modified with a text editor or if the file has been corrupted. In these cases, the parser cannot even read the RPL set to display it in the RPL set editor. If at any time, a RPL set is corrupted such that it cannot be opened, please contact RiverWare technical support at riverware-support@colorado.edu.
- **Printable**: A printable RPL set is one which can be read by the parser and displayed in the RPL set editor. This means that the expressions are syntactically correct, even though some may evaluate to data types which are inconsistent with their use in an outer scope. This level of validation is most frequently caused by an ambiguous expression being saved in the RPL set. Ambiguous expressions are expressions whose operator precedence allows more than one interpretation. Ambiguous expressions can be fixed by placing parentheses around the appropriate sub-expressions. A button for adding parentheses can be found in the rule palette. When RPL sets which only pass the printable validation are opened, errors are printed to the Diagnostics Output Window. The errors may then be fixed through the RPL set editor.
- **Consistent**: A consistent RPL set is one which is printable and whose expressions' data types are consistent with each other. For example, a sub-expression of type NUMERIC exists where a NUMERIC expression is expected in a higher level expression. Some expressions may be unspecified if they are of the correct data type. The consistency of data types is determined for all types except LIST, whose member types cannot be fully determined until the expression is evaluated at run time. A consistent RPL set may not properly evaluate at run time, but it will not generate any errors when it is opened.
- **Evaluatable**: An evaluatable RPL set is one which, as far as can be determined without actually evaluating it, could be successfully evaluated. This means that all of the previous validity levels have been satisfied. This also means that there cannot be any remaining unspecified expressions in the RPL set and that any object or slot references must map to an existing object or slot in the currently loaded model.

Given those definitions for the various results of validating a RPL set, the following actions can lead to validation errors.

Open RPL set: When a RPL set is opened, its functions and blocks are parsed from the file and the set is validated to the printable level. The printable validation ensures that the RPL set can be displayed in the RPL set editor's graphical user interface. RPL sets are also validated to the same level when they are saved to ensure that they can be reloaded at a later time. Some older RPL sets which contain critical errors, and corrupted RPL set files may fail to load. In these cases, a window appears indicating the location within the file where the error was detected.



A message will also appear in the Diagnostics Output Window indicating that the **RPL set loading failed:** and provide the path and name of the RPL set. The Diagnostics Output Window will display the line number in which the parsing error occurred. If this should occur, first verify that the file you are attempting to load is indeed a RPL set, then contact RiverWare technical support.

Load RPL set or Validate RPL set: When a RPL set is loaded by clicking on the **RPL Set Not Loaded** button in the RPL set editor or **Check Validity** is selected from the **RPL set** menu, the RPL set is validated to the evaluatable level. This means that all of the rule and function expressions are syntactically correct, have consistent expression data types, are fully specified, and reference objects and slots which exist on the workspace. If any of these criteria are not met, the RPL set is not validated and it cannot be loaded into the model. In this case, a window appears indicating one of the two messages:

Another message appears in the Diagnostics Output Window describing the validation error(s). Loaded RPL sets are also validated to the evaluatable level when a run is begun. This is to validate any changes which were made to the RPL set since it was successfully loaded. If any errors are detected, the run is stopped, and the appropriate diagnostic messages are displayed.

1.2 Evaluation and Runtime Errors

Several types of errors can be generated when evaluating RPL expressions. Some errors stop the run execution immediately, while others cause the current block to end with an early termination, but do not stop the run. The approach to debugging these errors varies depending on the error type. The error types are:

1.2.1 Non-Fatal RPL Evaluation Errors

These are errors which occur within the evaluation of a rule. Most often, these errors are produced when an engineering predefined function fails because it is attempting to model something physically impossible. These errors are often the result of the object state; i.e., the values in the object's slots. It is entirely possible for an engineering predefined function to fail during one rule evaluation, then succeed during another. The result of the engineering predefined function is almost always related to the state of the object at the time that the function evaluates.

For example, attempting to solve for the ending elevation of a reservoir given a starting elevation and flows which would cause overtopping of the reservoir would generate this kind of error. Some predefined functions can also fail in this way if they are attempting to access missing data or attempting to access data with incorrect arguments. Attempting to lookup values which do not exist in a tableslot would also generate a non-fatal evaluation error.

When the Rule Processor encounters a non-fatal rule evaluation error, it cannot continue evaluating the rule and must terminate early. Since the rule does not complete, no slots are set in the model. All existing slot values and priorities are still correct, and it is not necessary to stop the run.

When a non-fatal evaluation error is encountered, the error message is posted to the Diagnostics Output Window and immediately aborts the RPL statement. The block's dependencies are preserved, such that the block may be re-evaluated if any of its dependant slots change.

Example:

Consider the following rule which attempts to set Pool Elevation such that a target Storage is met. The **Target Storage()** is computed in an internal function. The **StorageToElevation()** predefined function is used to convert this target Storage to a Pool Elevation.

```
Res.Pool Elevation [] = StorageToElevation( %"Res", TargetStorage(),
    @"Current Timestep" )
```

If the **TargetStorage()** user function evaluated to a value of 15,000,000 acre-ft, and the ElevationVolumeTable of Res contains the following data:

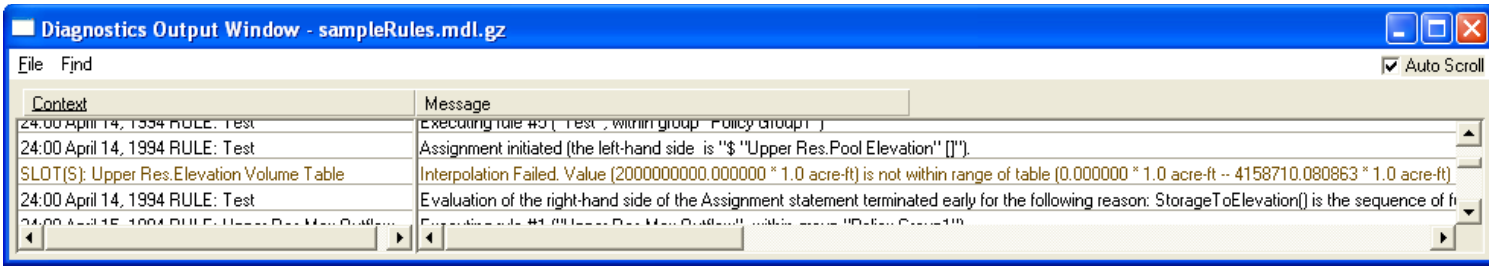
Pool Elevation	Storage
895.00	0
920.00 . . . 1060.00	811,000 . . . 8,241,000
1085.00	10,233,000
1110.00	12,452,000
1135.00	14,920,000

The predefined function, **StorageToElevation(),** will:

- Check the validity of the arguments:
  Res is an object on the workspace which contains an Elevation Volume Table.
  15,000,000 [acre-ft] is a numeric value in the unit type of volume.
  @"Current Timestep" is a valid timestep of the current model run.
- Attempt to find 15,000,000 [acre-ft] in the Storage column of the table.
- Fail to find 15,000,000 [acre-ft] in the Storage column:
  Post an error to the Diagnostics Output Window.
  Notify the rule that the evaluation of the predefined function failed.

Notice that there is nothing wrong with the **StorageToElevation()** function itself. The arguments which were provided to it are hypothetically valid. It just happens that the Storage value is too large for the data in the table. This is not a RPL set configuration or logic error. It is possible that the same rule could execute at a later time and that the **TargetStorage()** function would solve for a Storage of 14,000,000 [acre-ft]. Then, the **StorageToElevation()** function would succeed.

When the rule is notified that the **StorageToElevation()** function has failed with a data error, it cannot finish evaluating. The rule is aborted because there is incomplete information to continue.



Since the rule has not yet set any values in the model, the existing state of the model is still valid; it is based entirely on user input and assignments from successful rules. Since this failed rule has had no effect on the model, there is no need to stop the run. In addition, this rule's dependencies have been registered so that it can be re-fired if any of its dependant slots change at a later time. If Res' Storage or downstream demands were to change, this rule may re-fire and evaluate successfully.

Non-fatal RPL evaluation errors do not always indicate a problem with a RPL set. Although a red error message is displayed in the Diagnostics Output Window, the failure of the RPL expression may be a desired action. This may initially confuse users who are accustomed to seeing error messages only when the run has been aborted. Error messages which are generated for non-fatal rule evaluation errors are intended to give the modeler an indication of the failed functions even though the evaluation continues. These error messages should be investigated to determine whether the function failure really was desired.

Example:

In the previous example, the rule was attempting to set Res' Pool Elevation corresponding to a target storage. If the Pool Elevation is higher than the reservoir capacity, it would not be wise to set this value on the slot. This would not be a valid reservoir operation, and the ensuing dispatch would surely fail, aborting the run.

If this rule is part of a ruleset where the Pool Elevation or Outflow of Res may be set to meet several criteria, we may want the rule to fail. Consider the following ruleset:

Set Res Pool Elevation for Flood Control

Set Res Pool Elevation for Target Storage

Set Res Outflow for Surplus Conditions

Set Res Pool Elevation based on Guide Curve

If our Target Storage rule (#2) would result in a Pool Elevation which would overtop the reservoir, we may want to let the Surplus Conditions rule (#3) dictate the operation of the reservoir. Even though Rule #3 has a lower priority than Rule #2, we prefer its "safe" operation to Rule #2's. By evaluating an engineering predefined function which will "test" the mass balance of the reservoir, we give Rule #2 a chance to fail before it sets any slots. Once Rule #2 fails with the non-fatal error, the Rule Processor fires Rule #3. The design of this ruleset ensures that an unrealistic operation will not be attempted and that the run will not be aborted.

When examining non-fatal error messages, keep in mind that the failing function does not know whether its failure is critical or not. In a simulation dispatching context, a failed Elevation Volume Table interpolation is critical because it means the physical limit of the reservoir has been exceeded. In a rule evaluation context, a failed Elevation Volume Table interpolation is not very critical because it only means that a "what-if" calculation has exceeded the physical limit. In both cases, the table interpolation error is posted as soon as it is encountered. In the simulation dispatch case, the controller should immediately stop the run. In the rule evaluation case, the Rule Processor only needs to abort this rule and can then fire the next rule on the agenda.

1.2.2 Fatal RPL Evaluation Errors

Fatal RPL evaluation errors immediately abort execution. Unlike non-fatal RPL evaluation errors, these errors indicate a major problem with the RPL logic. These errors also occur within the evaluation of a RPL logic. Unlike non-fatal RPL evaluation errors, however, these errors are not dependant on the state of the system; they cannot be fixed by having different values in slots. These errors may be caused by missing arguments, arguments of the wrong data type, inconsistent unit types, invalid slot configurations, and/or missing objects and slots. These errors require intervention by the user if the run is ever to succeed. When the Rule Processor encounters a fatal rule evaluation error, the rule is aborted, the error message is posted to the Diagnostics Output Window, and the run is immediately stopped.

Predefined Functions: The number of arguments and data types of arguments to predefined functions are fixed. Each function checks the validity of its arguments when it is evaluated. The argument requirements for each predefined function are listed in RPL Predefined Functions section of the RiverWare help. If incorrect arguments are supplied to a predefined function, or the function fails to evaluate for other reasons, it may cause a fatal error. In this case, a message is posted in the Diagnostics Output Window and the run is immediately aborted.

Example:

The **ListSubbasin()** predefined function evaluates to a list of objects. The objects are the objects defined in the subbasin on the workspace. The function takes a single argument of type STRING. If the argument is not the name of an existing subbasin in the model, the function generates a fatal error.

Calling the function with a non-existent subbasin as below:

Print ListSubbasin ("Mane Stem of River")

The following error message is generated:

Evaluation of the Print statement failed for the following reason(s):
ListSubbasin() is the sequence of function calls (possibly containing only one function) which ended in an unsuccessful function call.
The function call failed for the following reason: Argument one is not a Subbasin.
This occurred at the following location within the expression:
"ListSubbasin ("Main Stem of River")".

Notice the structure of the error message. The first line indicates that the Print statement failed. The second and third lines indicate the reason the Print statement failed is that the evaluation of the **ListSubbasin()** function failed. The fourth line indicates that the function failed because the first argument is not a subbasin. The fifth and sixth lines indicate where in the rule the failed function call originated. Reading all of the error message is critical to understanding, and fixing, what went wrong.

Inconsistent Unit Types: The RPL processor automatically converts numeric values to the proper units for mathematical operations during RPL evaluation. As long as the unit type of a value is correct, its units can be converted without affecting the expression. If the dimensional analysis is incorrect, however, the RPL processor stops the run.

Example:

The mathematical expressions below may be evaluated because the unit types of their elements are consistent.

```
Res.Outflow []= 10 [cms]+  50 [cfs]
                FLOW=   FLOW+    FLOW

Res.Storage []= 50,000 [acre-ft/month]x  31 [day]
              VOLUME=             FLOWx   TIME
```

The values of expression elements on the right-hand side are converted to a common base unit prior to evaluation. Likewise, the result of the right-hand side is converted into RiverWare standard units so that it may be assigned to the slot.

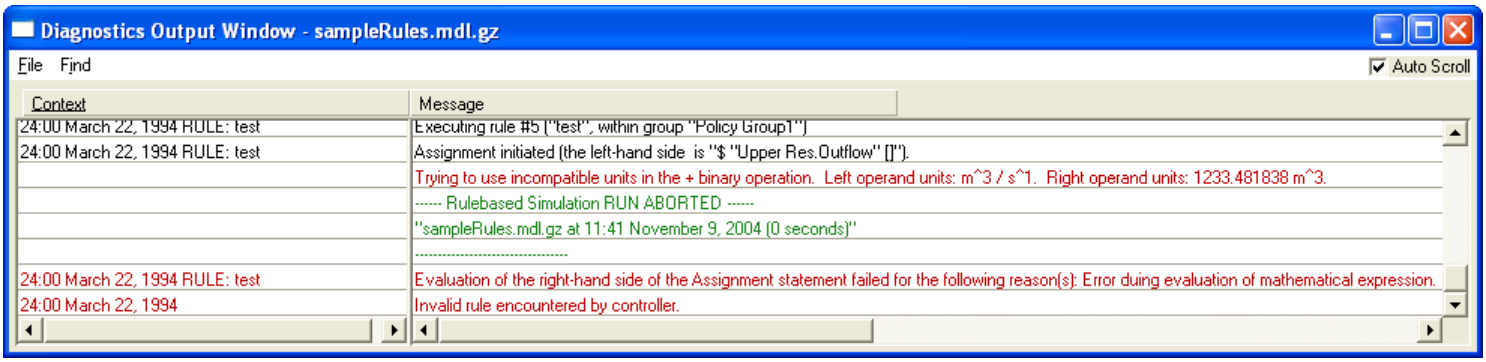
Example:

The mathematical expressions below may not be evaluated because the unit types of their elements are inconsistent.

```
Res.Outflow []= 10 [cms]+  50 [acre-ft]
                FLOW=   FLOW +    VOLUME

Res.Pool Elevation []= 50,000 [acre-ft/month]x  31 [day]
                LENGTH=             FLOW x   TIME
```

These expressions would cause fatal rule evaluation errors like the one below.



RPL unit type errors are only detected during evaluation.

Inconsistent Data Types: RPL requires that an expression evaluating to a particular data type be present where that data type is expected. This is enforced during the building of RPL expressions by only enabling expressions of the correct type in the Palette and by generating a parse error if an incorrect expression is directly typed into the text field. The error message, previously discussed, would read: "**The entered expression cannot replace the selected expression**."

Expressions of data type LIST are not verified during the building of RPL expressions. This is because a list may contain elements of any data type and because the number and type of list elements may change during the evaluation. As a result, inconsistent data types originating in lists will not be caught until the RPL expression or function is evaluated. These errors will stop evaluation immediately with an error.

Example:

The following rule will not generate any errors when it is constructed. When the ruleset is validated, it will be deemed "evaluatable."

PRINT { TRUE, 1"day" } < 0 > + @ "t"

When this rule is evaluated, however, it causes the run to abort with the following message:

Evaluation of the PRINT statement is invalid for the following reason(s): The left operand of + must be able to evaluate to a numeric or date/time value, and this is not currently possible. The problem was encountered at the following location within the expression: TRUE + t.

Here again, the message indicates that the addtion operation did not work, but the problem probably lies withing the list. Either access the next item in the list or change the 0th item to have a valid type.

Missing Objects or Slots: If an object or slot which is referenced in a RPL expression does not exist on the workspace when the RPL expression is evaluated, a fatal RPL evaluation is produced. This can occur when the slot or object name was improperly entered in the RPL expression, when a RPL set is

being used with the wrong model, or when object or data object slot names were changed after the RPL expression was created.

Names of objects and slots in a RPL set must exactly match an object and/or slot name on the workspace, including capitalization, spaces, and any underscores. Missing object or slot errors can be minimized by always using the Object Selector and Slot Selector to specify objects and slots in RPL expressions. Typing object or slot names directly into an expression textfield is not recommended due to the possibility of a syntax error.

Example:

The following rule will not generate any errors when it is constructed. When the ruleset is validated, it will be deemed "evaluatable."

Reservoir.Outflow[] = Reservoir.Inflow[]

When this rule is evaluated, however, it causes the run to abort with the following message:

Evaluation of the right-hand side of the Assignment statement failed for the following reason(s):
Failed to locate slot "Resarvoir.Inflow" on the global workspace.
This occurred at the following location within the expression: $"Resarvoir.Inflow".

1.2.3 Fatal Simulation Errors

These errors occur during the dispatching phase of a Rulebased Simulation run and only apply to Rulebased Simulation rulesets. These errors are produced when the controller priority and the priorities of slot values are such that objects cannot solve the model. These errors are usually caused by objects dispatching with an unintended dispatch method, excessive redispatching of an object, and/or an overdetermined multislot. When these occur, the Rulebased Simulation Controller cannot determine how to continue, and the run is aborted with an error.

Fatal Simulation errors are often the most complex errors to understand and debug. These errors occur during the simulation dispatching which takes place after a rule successfully sets a slot value. In order to simulate the effects of the new slot value, many objects may need to redispatch. The dispatch method with which each object redispatches is determined from the object's slot priorities. Occasionally, the priorities of the slots are such that a unique dispatch method cannot be identified. When this happens, the object will attempt to redispatch with the same method as it dispatched with the last time. One of two error situations may result.

Junior/Senior Slot Priority Error: If the slot which was just set by a rule is solved for during this dispatch, the slot priority conflict will cause the run to abort with an error. The error often indicates:

Attempting to set senior slot (*priority*) with a junior priority (*same priority*).
Assignment attempted within the dispatch triggered by rule #*same priority*.
The object probably did not dispatch with the intended method.

When this error occurs, the rule at the indicated priority is usually to blame. Often, this rule has set a slot value at a priority which confuses the simulation into choosing the wrong dispatch method.

Example:

Consider a Reach object called Rio. Rio's Inflow may be determined by the release of an upstream reservoir or its Outflow may be determined by downstream demands or environmental considerations. Both may be specified as long as the slot's priorities can be used to determine the preferred solution.

The ruleset used to control Rio is shown below:

1. Set Rio Inflow due to Minimum Upstream Reservoir Release
Rio.Inflow[] = IF (Rio.Inflow[] < DataObj.MinResRelease[]) THEN
DataObj.MinResRelease[]

2. Set Rio Outflow for Maximum Fish Flow
Rio.Outflow[] = IF (Rio.Outflow[] > DataObj.MaxFishFlow[]) THEN
DataObj.MaxFishFlow[]

3. Set Rio Outflow to Meet Downstream Demands
Rio.Outflow[] = DataObj.TotalDownstreamDemand[]

The Rulebased Simulation proceeds as follows:

- Initially, no values are known, so Rio cannot dispatch.
- Rule #1 fires but terminates early because Rio.Inflow is unknown.
- Rule #2 fires but terminates early because Rio.Outflow is unknown.
- Rule #3 fires and sets Rio.Outflow to meet demands.
- Rio dispatches with the SolveInflow dispatch method based on these priorities:
  Inflow = unknown
  Outflow = known at priority 3R
  and solves for Inflow at controller priority 3. This puts rules #1 and #2 back onto the Agenda.
- Rule #1 fires and sets Rio.Inflow for minimum upstream release (assuming the rule logic evaluates to TRUE). The rule can overwrite the existing priority 3 Inflow with the new priority 1R Inflow.
- Rio redispatches with the SolveOutflow dispatch method based on these priorities:
  Inflow = known at priority 1R
  Outflow = known at priority 3R
  and solves for Outflow at controller priority 1. Rule #2 is still on the Agenda.
- Rule #2 fires and sets Rio.Outflow for maximum fish flow (assuming the rule logic evaluates to TRUE). The rule can overwrite the existing priority 1 Outflow with the new priority 2R Outflow.
- Rio redispatches with the SolveOutflow dispatch method based on these priorities:
  Inflow = known at priority 1R
  Outflow = known at priority 2R
  and solves for Outflow at controller priority 2. This was not the intended dispatch! Rule #2 just set Rio.Outflow. The effect of this rule should not be to solve for a new Rio.Outflow. When the dispatch attempts to overwrite Outflow at priority 2R with a new value at priority 2, an error is gen-

erated.

The responsibility for the error, in this case, falls squarely on Rule #2. This rule should not be attempting to enforce a maximum fish flow when a higher priority minimum release rule is controlling the river. Unfortunately, Rule #2 does not know that Rule #1 is controlling the river. There are two solutions to this error.

Because the rules are acting on different slots, they cannot depend exclusively on priorities to determine the appropriate solution. If Rule #2 were attempting to set Rio.Inflow instead, the rule would fail gracefully during the rule execution. The existing priority 1R would prevent Rule #2 from overwriting the slot value. Of course, changing Rule #2 to set Inflow could now create a similar problem between Rule #2 and Rule #3.

The other solution is to make Rule #2 "smarter." Rule #2 could check the Inflow of Rio to see if it is at a level corresponding to the minimum upstream release. If this is the case, Rule #2 could then decide not to return a value from its right-hand side. It would exit ineffectually. Another way to let Rule #2 know that Rule #1 is controlling the river is to have Rule #1 set a "state flag" on a data object in the model. State flags are slots whose values indicate the current state of the system, such as surplus, shortage, or minimum release. A state flag could be checked by Rule #2 to decide if it should change Rio's Outflow.

Infinite Loop Dispatching Error: Another error situation which can result from incorrect dispatching is an infinite loop. This can occur between two objects which are both solving for the same slot. Each object successfully solves for a new value on the slot, overwriting the previous value. Each new slot assignment triggers the other object to redispatch, during which it also solves for a new value on the slot. This continues until the maximum iterations are reached on one of the objects' slots or the modeler terminates the RiverWare session.

Example:

Consider two Reach objects called UpperRio and LowerRio and a Diversion called UpperRioDiversion. The diversion is attached to the UpperRio Reach and can divert an amount of flow equal to the amount it leaves in the UpperRio. There are also irrigation demands downstream of the LowerRio. The system usually solves from the bottom up; downstream demands are set on LowerRio's Outflow, then the Diversion Request is set on the UpperRioDiversion. As in the previous example, UpperRio's Inflow may be overwritten by a minimum release rule.

The ruleset used to control the Rios is shown below:

1. Set UpperRio Inflow due to Minimum Upstream Reservoir Release
   UpperRio.Inflow[] = IF (UpperRio.Inflow[] < DataObj.MinResRelease[]) THEN
   DataObj.MinResRelease[]

2. Set UpperRioDiversion Not to Exceed UpperRio.Outflow
   UpperRioDiversion.Diversion Request[] = MIN (DataObj.UpperRioDivRequest[],
   UpperRio.Outflow[])

3. Set LowerRio Outflow to Meet Downstream Demands
LowerRio.Outflow[] = DataObj.TotalDownstreamDemand[]

The Rulebased Simulation proceeds as follows:

- Initially, no values are known, so neither Rio can dispatch.
- Rule #1 fires but terminates early because UpperRio.Inflow is unknown.
- Rule #2 fires but terminates early because UpperRio.Outflow is unknown.
- Rule #3 fires and sets LowerRio.Outflow to meet demands.
- LowerRio dispatches with the SolveInflow dispatch method based on these priorities:
  Inflow = unknown
  Outflow = known at priority 3R
  and solves for Inflow at controller priority 3. This propagates across the link to UpperRio.Outflow, causing Rule #2 to be put back onto the Agenda.
- UpperRio dispatches with the SolveInflow dispatch method based on these priorities:
  Inflow = unknown
  Outflow = known at priority 3
  but cannot solve completely because the UpperRioDiversion is not yet known.
- Rule #2 fires and sets the UpperRioDiversion's Diversion Request equal to the UpperRio Outflow (assuming that the request in the Data Obj is very large).
- UpperRioDiversion dispatches and sets the Diversion from UpperRio at controller priority 2. This propagates across the link to UpperRio.Diversion.
- UpperRio redispatches with the SolveInflow dispatch method based on these priorities:
  Inflow = unknown
  Outflow = known at priority 3
  Diversion = known at priority 2
  and solves for UpperRio.Inflow at controller priority 2. This causes Rule #1 to be put back onto the Agenda.
- Rule #1 fires and sets UpperRio.Inflow for minimum upstream release (assuming the rule logic evaluates to TRUE). The rule can overwrite the existing priority 2 Inflow with the new priority 1R Inflow.
- UpperRio redispatches with the SolveOutflow dispatch method based on these priorities:
  Inflow = known at priority 1R
  Outflow = known at priority 3
  Diversion = known at priority 2
  and solves for UpperRio.Outflow at controller priority 1. This puts Rule #2 back on the Agenda.
- LowerRio redispatches with the SolveOutflow dispatch method based on these priorities:
  Inflow = known at priority 1
  Outflow = known at priority 3R
  and solves for Outflow at controller priority 1.

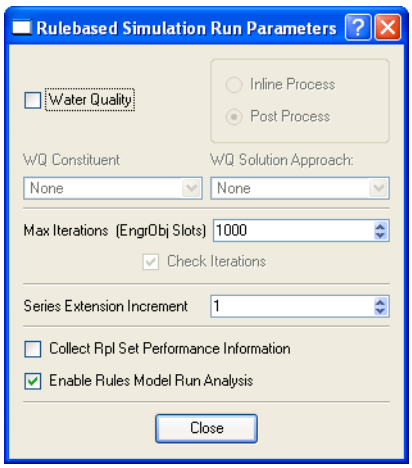
Now the stage is set for the error to take over.

- Rule #2 fires and resets the UpperRioDiversion's Diversion Request equal to the new UpperRio

Outflow (assuming, again, that the request in the Data Obj is very large). The rule can overwrite the existing priority 2R Diversion Request with a new priority 2R Diversion Request.

- UpperRio redispatches with the SolveOutflow dispatch method based on these priorities:
  Inflow = known at priority 1R
  Outflow = known at priority 3
  Diversion = known at priority 2
  and solves for a new UpperRio.Outflow at controller priority 2. The dispatch can overwrite the existing priority 1 Outflow with a new priority 2 Outflow. This value propagates across the link to LowerRio's Outflow. This also causes Rule #2 to be put back on the Agenda.
- LowerRio redispatches with the SolveInflow dispatch method based on these priorities:
  Inflow = known at priority 2
  Outflow = known at priority 1
  and solves for a new Inflow at controller priority 2. This propagates across the link to Upper-Rio.Outflow.
- UpperRio redispatches again with the SolveOutflow dispatch method based on these priorities:
  Inflow = known at priority 1R
  Outflow = known at priority 2
  Diversion = known at priority 2
  and solves for a new UpperRio.Outflow at controller priority 2. This value propagates across the link to LowerRio's Outflow.
- The two Rios continue redispatching with the wrong dispatch methods and overwriting the results of each other's solutions.

Slot maximum iteration checking is turned on by default and cannot be turned off when performing Rulebased Simulation, though the number of maximum iterations can be changed by the user. This iteration default can be viewed as a grayed box for "Check Iterations" in the **Rulebased Simulation Run Parameters** dialog, invoked from the **View** menu of the **Run Control** dialog. Within this dialog, the user can change the number of maximum iterations (the default number of maximum iterations is 20). For more complex models, the user may be required to increase the number of maximum iterations.



1.2.4 Fatal Rulebased Simulation Errors

These errors occur when a rule is fired too many times within any timestep. Rules may not be evaluated more than 50 times per timestep. A rule firing more than 50 times is usually an indicator of a circularity in the rule logic. Excessive rule firing and redispatching is an indication of a circularity in the ruleset's design; one or more rules are dependent on the slots which they are setting. Even in the most complex rulesets, individual rules should not fire more than a few times in any given timestep. Unchecked, circularities would continue indefinitely, or until the user terminates the executable. But, when a rule is fired more than 50 times, the Rulebased Simulation

Controller stops the run and posts this error to the Diagnostics Output Window: “Max rule executions exceeded for timestep.” Luckily, these types of circularities are rare.

Example:

Circularities in rulesets are easy to create consciously but they rarely happen by accident.

An example of a circular ruleset which causes no dispatching, is shown below:

```
.Data.A[] = IF ( Data.B[] == 1 [NONE] ) THEN
   0 [NONE]
   ELSE
   1 [NONE]

Data.B[] = IF ( Data.A[] == 1 [NONE] ) THEN
      1 [NONE]
   ELSE
      0 [NONE]

Data.A[] = 1 [NONE]
```

The Rulebased Simulation would proceed as follows:

- Rule #1 fires but terminates early because Data.B is not known.
- Rule #2 fires but terminates early because Data.A is not known.
- Rule #3 fires and sets Data.A to 1. Rule #2 goes back on the Agenda.
- Rule #2 fires and sets Data.B to 1. Rule #1 goes back on the Agenda.
- Rule #1 fires and sets Data.A to 0. Rule #2 goes back on the Agenda.
- Rule #2 fires and sets Data.B to 0. Rule #1 goes back on the Agenda.
- Rule #1 fires and sets Data.A to 1. Rule #2 goes back on the Agenda.
- The steps above are repeated another 24 times before the Rulebased Simulation Controller stops the run.

1.3 Understanding RPL Evaluation

RiverWare provides the following tools for helping the user to understand RPL evaluation and its consequences in rules, goals, methods, and expression slots:

- Debugging: The user is able to pause execution, look at the values of RPL expressions as they happen, and then step through RPL expressions.
- Diagnostics: Print out messages when rules or functions are executed and evaluated. Several categories are especially relevant to the execution of RPL policy, including “Rule execution”, “Function execution”, and well placed “Print” statement).
- The rulebased simulation **Model Run Analysis** tool: provides information on the rules and policy that caused an object to dispatch. This tool is described **HERE (ModelRunAnalysis.pdf, Section 2)**
- The **RPL Set Analysis Tool**: provides static analysis only, i.e., analysis which is not specific to a particular run. This tool is described in depth **HERE (Section 5)**.

With these tools it is possible to examine RPL behavior and make changes if that behavior is deemed to be unintended. For complex policies this process of policy debugging can be very time consuming. This document describes Debugging and the RPL Set Analysis Tool and provides links to documentation of diagnostics and the model run analysis tool.

RPL Debugger

2. RPL Debugger 

RPL sets can be very large and complicated, and consequently it can be difficult to determine why a set is doing what it is doing. The RPL debugger is designed to help understand the behavior of RPL sets; with this tool the user can pause RPL execution, look at the values of RPL expressions as they are evaluated, and step through RPL execution. The following sections describe the RPL debugger including an overview, how to enable debugging, a tour of the debugger dialog, and suggested ways of using the debugger.

2.1 Overview of RPL Debugging

The RPL Debugger utility provides the following functionality:

- Control execution at a fine granularity. Examples:
  - Pause (interrupt) execution.
  - Set a breakpoint which pauses execution.
  - Continue execution.
  - Step, i.e., execute the next line of code, optionally descending into called functions.
- Visualize the currently executing source code and data when the program is interrupted, either because an error was encountered or because the user paused execution through the debugger. The RPL policy editors provide a debug cursor which provide visual indication of the current line (i.e., the next line to be executed) and a mechanism for displaying a textual representation of the current values of expressions.
- Visualize and traverse the call stack when the program is interrupted. This consists of a list of the names of the blocks and functions currently in the call stack with an indication of the current “location” in that stack.
- Manage collections of breakpoints, i.e., provide operations such as add, delete, temporarily disable.

RPL is used in several contexts within RiverWare; currently they are:

- Rulebased simulation (RBS) - Ruleset
- Object level user defined accounting methods - Method set
- Initialization Rule - Ruleset
- Iterative MRM - Ruleset
- Optimization Policy - Goal set
- Expression slots

Except for this last application (Expression slots), RPL policy is organized at the RPL group level into functions and blocks of RPL statements, variously called rules, goals, or methods. Statements can be nested and contain expressions, which might contain function calls. Expression slots and functions are defined by a single RPL expression.

Thus, for blocks of statements, the basic unit of execution in RPL is the statement; whereas, for functions and expression slots, it is the single defining expression.

This RPL policy organization is reflected in the locations at which execution can be paused:

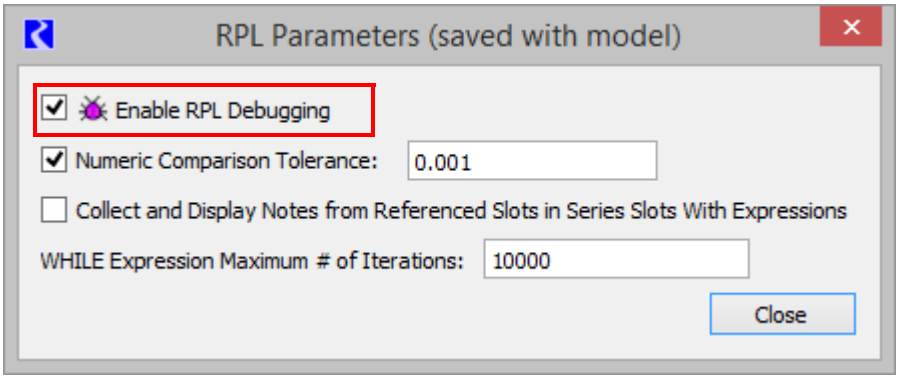
- Before a statement in a rule, optimization goal, or user-defined accounting method is executed
- After all statements in a rule, optimization goal, or user-defined accounting method have executed
- Before a function's body expression is evaluated
- After a function's body expression is evaluated
- Before an expression slot's expression has evaluated
- After an expression slot's expression has evaluated

When execution is paused, the debugger allows the user to view the values to which expressions have evaluated up to that point. Each time a block of statements or an expression slot is executed, results from any previous evaluations are cleared, so one must pause after a statement's execution to see the values to which its expressions last evaluated.

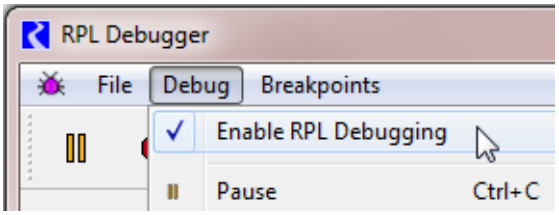
2.2 Enabling RPL Debugging

To enable this feature, use one of the following approaches:

- In the RPL Parameter dialog (**Policy ➥ RPL Parameters**), click the **Enable RPL debugging** toggle as shown to the right.

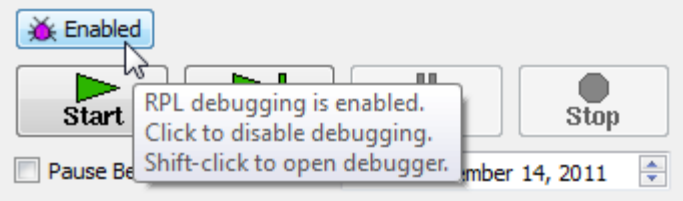


- From the RPL Debugger (**Policy ➥ RPL Debugger**), use the **Debug ➥ Enable RPL Debugging** menu toggle as shown below.



This is a workspace-level toggle and applies to the execution of all RPL policy associated with the current model. This setting is saved with the model.

When debugging is first enabled, a button is added to the Run Control dialog to indicate the state of the RPL Debugger. Click the button to toggle the state. Shift-Click the button to open the RPL Debugger. This button remains in place for that RiverWare session. To hide the button, use the **View ➥ Show RPL Debugging Button**.



Note: Enabling debugging can have a significant negative impact on performance!

There are two distinct performance impacts of debugging:

- **Retaining values on expressions** - Normally, RPL execution is optimized for efficiency, and thus during execution values are reused where possible and intermediate results are not retained. To do so requires additional memory and time to copy intermediate results.
- **Computational overhead** - To support the interruption of RPL execution at each point at which a pause might occur, RiverWare must check if a pause is in fact appropriate for that location.

Since both of these impacts can be significant, by default RiverWare does not enable these processes. Rather the user only incurs the additional overhead of debugging when they have indicated that they would like interactive debugging as described above.

A scenario in which the user might want to disable and then enable debugging temporarily is presented HERE (Section 2.5.1).

2.3 Tour of the RPL Debugger Dialog

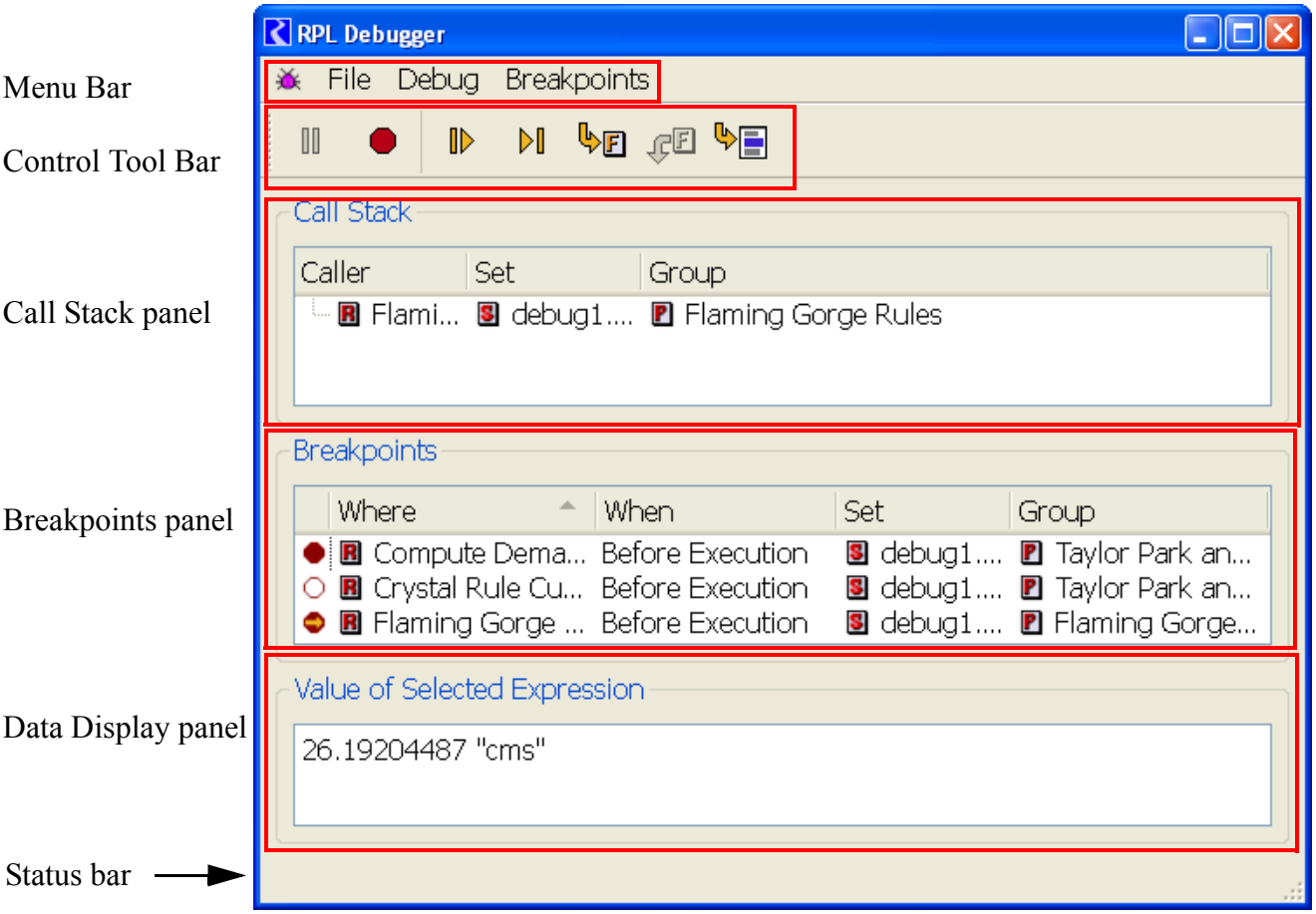
The RPL Debugger dialog is accessible from the:

- Workspace **Policy** menu (**Policy ➥ RPL Debugger**)
- From any RPL dialog's **Block** (**Set, Rule, Goal, Method**, **Function**, etc.) **➥ Debugger** menu
- Using the F8 shortcut when any RPL dialog is selected

This dialog displays information about RPL execution and allows the user to control RPL execution. The principle components of this dialog are as follows and shown in the screenshot:

- Menu bar: menu access to debugger functionality.
- Control tool bar: buttons for controlling RPL execution.
- Call Stack panel: describes the location at which RPL execution is paused (when it is paused).
- Breakpoints panel: lists the locations at which the user has requested that execution regularly pause.
- Data Display panel: displays the value to which the selected expression last evaluated.
- Error message panel: when an error occurs during RPL execution, a description of that error. (described HERE (Section 2.4.7))

- Status bar: brief description of the state of the debugger.



2.3.1 Menu Bar

The following table describes the functionality available through the **File** and **Breakpoints** menus. The functionality of the **Debug** menu is described in the next section **HERE (Section 2.3.2)**.

Menu	Sub-menu	Keyboard accelerator	Description
File	Close	Ctrl+w	Close the RPL Debugger
Breakpoints	Enable Breakpoint	F9	Enable (or disable) the selected breakpoint
Breakpoints	Delete Breakpoint	Shift+F9	Delete the selected breakpoint
Breakpoints	Delete All Breakpoints	Ctrl+Shift+F9	Delete All Breakpoints

2.3.2 Control Tool Bar and Debug Menu

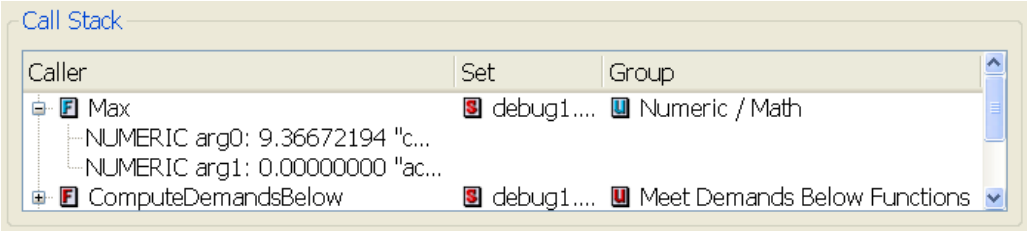
In addition to the existing mechanisms for controlling a run (e.g., the pause button of the run control dialog), through the debugger the user can control RPL execution. The control toolbar and **Debug** menu of the debugger dialog provides the following button controls:

Menu Item	Button	Keyboard accelerator	Description
Enable RPL Debugging			Turn on or off the collection of RPL Debugging information as described HERE (Section 2.2).
Pause		Ctrl+c	During RPL evaluation, pause at the next opportunity. When RPL execution is paused, the user can visualize the state of the system in the following ways: • Examine the value to which any RPL expression last evaluated. • Examine the current call stack. • Examine the workspace or otherwise interact with any of the RiverWare dialogs. Some of this functionality is available as well after a run has terminated. The user is strongly discouraged from making any changes to the model or policy (and is prevented from exiting RiverWare) while RPL execution is paused.
Stop		Shift+F5	Abort RPL execution. If in a run, stop the run at the next available opportunity. The user is not presented with the usual abort run notification message but a green diagnostic is posted.
Continue		F5	When execution is paused, continue execution until one of the following occurs: • A breakpoint is reached. • The user requests another pause or stop event. • RPL execution terminates normally.
Step		F10	When execution is paused, continue execution and pause at the next legal spot for pausing execution which is at an equal or lesser depth in the call stack. If paused at a statement, this will continue to the next statement.
Step Into		F11	When execution is paused, continue execution and pause at the next legal spot for pausing execution including any functions called.

Menu Item	Button	Keyboard accelerator	Description
Step Out		Shift+F11	When execution is paused within a function, continue execution and pause at the next legal spot for pausing execution which is at a lesser depth in the call stack (i.e., continue to the next pausable location in the calling function).
Cont. to Sel.		F12	When execution is paused, continue execution until the selected RPL expression is about to be executed (or execution is interrupted for some other reason, e.g., another breakpoint). This action is functionally equivalent to: set breakpoint before selection, continue, remove breakpoint
Show Button Labels			Show the labels on the toolbar buttons. (shown above)

The control toolbar can be repositioned within the dialog or detached and displayed outside of the dialog. By default, the buttons are unlabeled, but text labels can be displayed by selecting **Debug ➥ Show Button Labels**. All of the control actions are also available via the Debug menu or by keyboard shortcuts. For the most part, the shortcuts are the same as those used by Microsoft Visual Studio for the comparable action.

2.3.3 Call Stack Panel



When RPL execution is paused in the debugger, the Call Stack panel describes the current execution location. The display is a treeview list of the items which have are currently executing. Double-clicking an item in a row in this list opens the associated editor. For each item in the call chain, its **Caller**, **Group**, and **Set** are listed. For functions, the argument types, names, and values are displayed as optionally displayed children of that row.

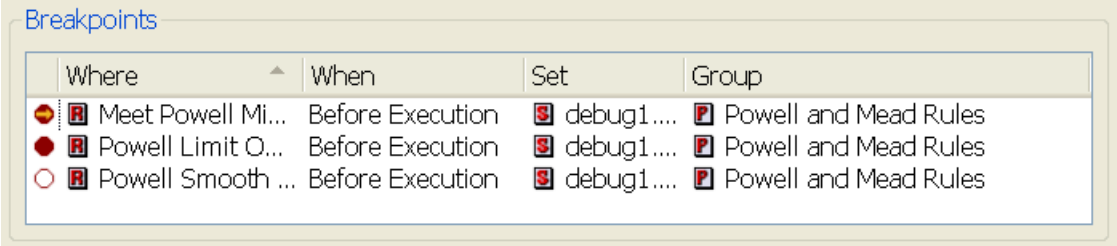
2.3.4 Breakpoints Panel

The Breakpoints panel lists locations at which the user has requested that execution regularly pause. Double clicking on a row opens the editor for that item. For each breakpoint, the panel displays:

- **Icon**: a red octagonal breakpoint indicator, filled indicates that it is enabled ●, unfilled indicates disabled ○. Left clicking on the stop sign indicator will enable/disable the breakpoint.

- **Where**: the name of the item containing the breakpoint. The breakpoint name includes an identifier which indicates the statement with which the breakpoint is associated. It is a hierarchical ID which looks like: <block/rule priority>.<statement index>.<statement index>. ...

  E.g., for the screenshot below which has two statements, for the first statement in rule 9, the **Where** column says: **Green Valley Diversions (9.1)**
- **When**: if it will break before or after execution of that item.
- **Group**: the group (where applicable) that contains that item.
- **Set**: the set (where applicable) that contains that item.



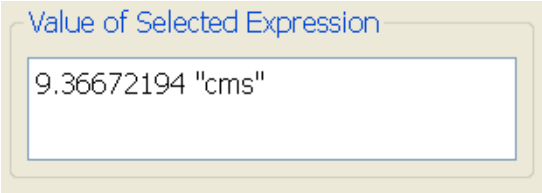
The list may be sorted based on any column; initially it is sorted based on the name of the item containing the breakpoint. Also the columns may be rearranged by dragging the column label.

When RPL execution pauses at a breakpoint, the associated row in the Breakpoints Panel is scrolled into view. While execution is paused at a breakpoint, the breakpoint indicator contains a yellow arrow (the debug cursor) .

Breakpoints can be created and deleted within the RPL editor for the corresponding item **HERE (Section 2.5)**. Within the RPL Debugger dialog, they can be deleted and/or temporarily disabled/enabled using the Breakpoints menu. Also, left clicking on the stop sign indicator will enable/disable the breakpoint. Breakpoints are persistent (saved with the RPL item which contains the breakpoint).

2.3.5 Value of Selected Expression Panel

When paused or after a run, the **Value of Selected Expression** panel displays the value of the selected RPL expression. This is described in more detail **HERE (Section 2.4.3)**.



2.4 Using the RPL Debugger

Following is a description of how to use the RPL Debugger. In general, the approach is: add breakpoints, start execution, execute to the breakpoint and pause, investigate values at that breakpoint, continue or step to the next location of interest, and continue looking at values and stepping/continue until satisfied. The process becomes more complicated when you wish to investigate the values at one timestep or location within a long run; this scenario is described **HERE (Section 2.5)**.

2.4.1 Adding breakpoints to RPL dialogs

As discussed earlier, RPL execution can be paused at the following locations:

- Before a statement in a rule, optimization goal, or user-defined accounting method is executed
- After all statements in a rule, optimization goal, or user-defined accounting method have executed
- Before a function's body expression is evaluated
- After a function's body expression is evaluated
- Before an expression slot's expression has evaluated
- After an expression slot's expression has evaluated

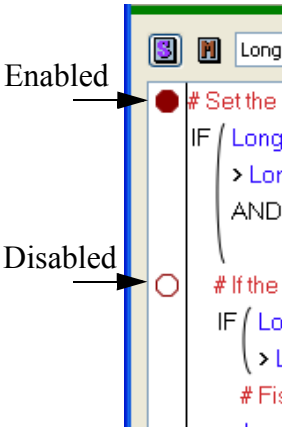
Setting a breakpoint at a particular location will cause execution to pause each time that location is reached. When debugging is enabled, RPL dialogs displaying a location at which RPL execution might pause (rule, function and expression slot dialogs) display a margin on the left in which debugging indicators are drawn. RPL frames in which it is not possible to pause RPL execution (e.g., rule execution constraint frames) never show a margin for debugging indicators.

Left clicking in the margin adds a breakpoint ("break before") at that location if there is not one already; disables it if there is an enabled one, and deletes it if there is a disabled breakpoint. That is, clicking the margin cycles through three states, enabled, disabled, and deleted. If the click location is below the statement or functions, then a "break after" breakpoint is added at that location.

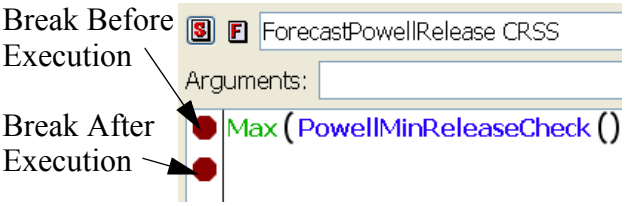
Breakpoints may also be added/removed from the selected expression using the **Rule/Methods/Goal(block) ➥ Break Before/After Execution** menu actions. When a RPL expression is selected, it is only possible to add a break point before that selection. In this case, RiverWare adds the breakpoint before the **statement** (assignment, if, ForEach, etc...) containing the selection. Otherwise, for the entire block, "Break Before" applies to the first statement and "Break After" applies to the last statement. The **Break Before/After Execution** menu items of the RPL editor dialogs are enabled and toggled based upon the RPL selection and contents of the dialog (as well as whether or not debugging is enabled).

Note: The debugging indicators are sized to match the font, which may be changed in the RPL Layout dialog.

If there is a breakpoint associated with a given location, a solid red octagon is drawn in the debugging margin (unfilled red octagon for disabled breakpoints). For blocks, the breakpoint indicator is associated with a specific statement or appears just below the last statement (indicating a breakpoint which is activated after the last statement has executed).



For functions, the breakpoint indicator appears in the function editor and appears at the top of the expression body for a "before execution" breakpoint and just below the body for an "after execution" breakpoint.



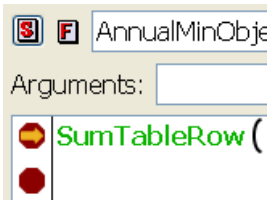
2.4.2 Starting the debugger

To start the debugger, initiate execution in the standard way depending on the set to be debugged. That is:

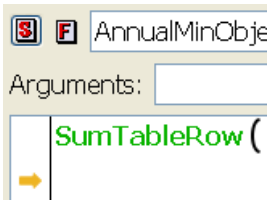
- Rulebased simulation ruleset: Start a rulebased simulation run.
- Optimization policy set: Start an optimization run.
- Object level user defined accounting method set: Start an accounting run.
- Initialization Rules: Start a simulation or rulebased simulation run.
- Iterative MRM Ruleset: Start an iterative MRM run.
- Expression slots: Start a run or evaluate the expression slot(s) manually.

The RPL execution will begin and the RPL debugger will pause when it hits a breakpoint or is paused.

When RPL execution is paused, a yellow arrow called the *debug cursor* ➡, is shown in the RPL dialog margin to the left of the statement or expression at which execution is paused. Typically this indicates that the indicated statement/expression is just about to be evaluated. Note that when paused after execution of a function or the last statement of a block, the debug cursor will not be pointing to a specific function/statement but will appear at the bottom of the dialog.



Debug cursor in a function at Break Before Execution

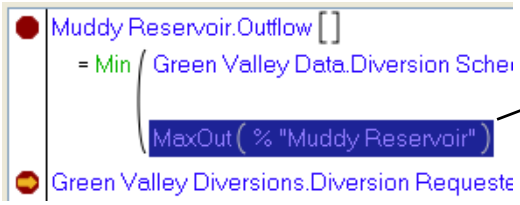


Debug cursor after function execution (no breakpoint)

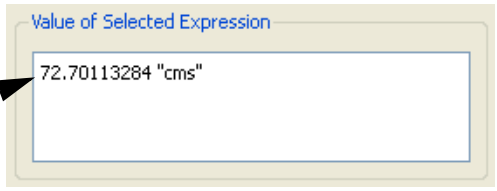
2.4.3 Displaying Data Values

When RPL execution is paused in the debugger, the user can interact with most RiverWare dialogs to examine the current state of the system. The values of interest during a run are the values to which expressions (including sub-expressions, i.e., expressions within expressions) have evaluated, as well as the values assigned by assignment statements. In many debuggers, the user enters the variable for which they want to see the value. Since RPL expressions are not named, the user can not pick an expression for display by entering its name or by selecting it in a list of expression names; rather the most straightforward method is to select the expression in the RPL editor in which it is displayed. Once selected, there are two ways to look at the value of the selected RPL expression:

- using the **Value of Selected Expression** panel in the debugger

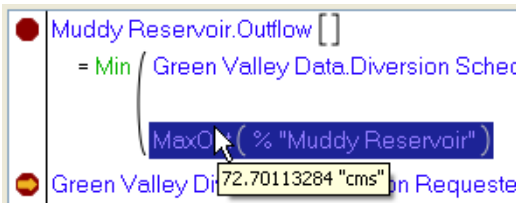


RPL dialog with Selection



Debugger dialog displaying value

- using tooltips on RPL dialogs. Tooltips are shown by hovering over the selected expression.



RPL dialog with tooltip showing value

2.4.4 Data Value Units

The value displayed uses the settings in the Unit Schemes **HERE (Units.pdf, Section 2)**. Whenever a RPL value is displayed in the debugger or in tooltips, RiverWare uses the units defined in the scheme. But, for monthly and annual values, it is sometimes not known within the debugger for which timestep the value is associated. In that case, monthly values assume there are 31days in the month. Annual values assume 365days in the year.

2.4.5 Stepping, Continuing, and Pausing

Once paused, the user can investigate the values of expressions and then **continue** on to other breakpoints, **step**, **step into**, **step out**, or **continue to selection**. These actions are available on the RPL debugger toolbar, menu choices, and keyboard accelerators as described **HERE (Section 2.3.2)**.

Also remember that within a run, there are two types of stepping and pausing:

- From the RPL debugger and
- From the Run Control dialog.

For example, within a rulebased simulation run, you click Step on the Run Control to advance the timestep, use the RPL debugger to break in a rule, step through that rule in the RPL debugger and then click continue on the RPL debugger. The timestep then finishes and the Run Control will pause before the next timestep.

2.4.6 Limitations

There are a few limitations that are imposed by the debugger:

1. Because you are in the middle of executing a RPL item, it would be problematic to delete an object or close RiverWare. Thus, when RPL execution is paused in the debugger, RiverWare prohibits many user actions, such as:

- Exiting RiverWare
- Closing a RPL set
- Clearing the workspace
- Loading a model

2. The debugger will not display the value of the entire left hand side of an assignment statement. Technically, the left hand side is not evaluated, it receives the value to which the right hand side evaluates, so it does not have a value to display. If you wish to see the value that will be assigned, highlight the entire right hand side. Note sub-expressions on the left hand side of an assignment are evaluated, and are available for display (after the statement has been executed).

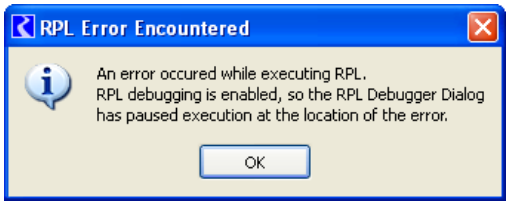
3. As discussed earlier, enabling debugging incurs performance costs in both memory and CPU usage. The exact impact is highly model- and machine- dependent, but most user can expect to see less than a 25% slowdown in RPL execution time.

2.4.7 Error Handling

If an error occurs during RPL evaluation with debugging enabled, the user is presented with an informational dialog and execution is paused in the debugger. The RPL editor containing the location of the error is raised and the debugging cursor indicates the location of the error. This could be one of the following:

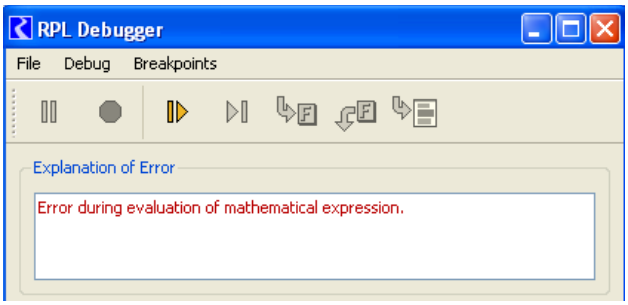
- A statement in a block
- The body of a function
- The expression of an expression slot.

RPL execution is paused as soon as the error occurs, and an informational dialog is presented to the user. This happens even before an error has been posted to diagnostics, however all of the relevant information concerning the error is displayed within the debugging dialog:



- The call stack contains the items being evaluated at the time of the error

- A panel is displayed with the label “Explanation of Error” which provides an explanation of the error and a textual version of the RPL expression at which the error occurred. This panel is hidden unless there has been an error.



The only action possible through the debugger when the run has been aborted is to continue execution, which will post the error diagnostic and abort the run as usual. 

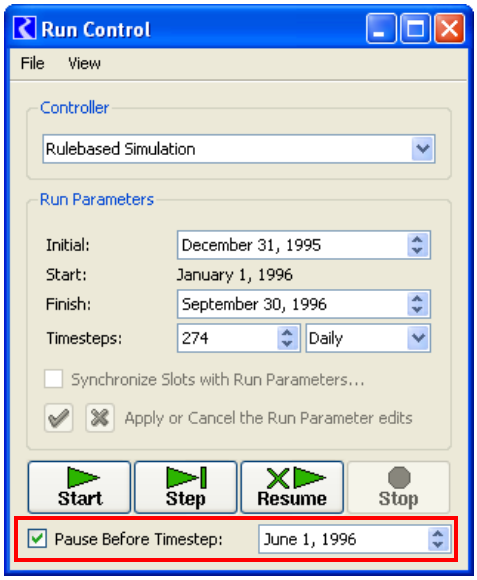
2.5 Sample Use Scenarios

In this section we present a few example scenarios that illustrate different ways to make use of the RPL debugger.

2.5.1 Scenario 1

The Run Control dialog allows user to specify a timestep/goal at which to pause the run. This is useful for longer model runs when you wish to look at execution on a specific timestep. The following is a possible debugging scenario:

- Disable RPL debugging as described **HERE (Section 2.2)**.
- Configure the Run Control to pause before the desired date as shown in the screenshot.
- Start a run and let it run until it pauses at the timestep indicated
- Enable RPL debugging.
- Set up breakpoints as needed. This could actually be done at any point. In the initial part of the run, RPL debugging is disabled, so the breakpoints have no effect.
- Continue or step the run from the Run Control.
- The run proceeds until a breakpoint is reached.
- Examine RPL values as desired and use the debugger’s Step/Continue, buttons to control RPL execution.
- When finished, delete or disable breakpoints and continue in the RPL debugger and in the Run Control to finish. Or, click the Stop button in the debugger to stop RPL execution and end the run.



2.5.2 Scenario 2

Because the debugger catches RPL errors and displays the exact location of the error, it can be used in the RPL development process to catch run time errors. For example, when running a model where you just made a lot of RPL changes and you wish to catch runtime errors:

- Enable debugging but do not set any breakpoints.
- Run the model or execute the policy.
- The model runs, and if a RPL error occurs the RPL debugger will pause execution.
- The debugger will display an explanation of the error and open the dialog indicating the location of the error, as described **HERE (Section 2.4.7)**.
- Determine what the error is by using debugging tools.
- Click **Continue** in the debugger and the run will abort.
- Fix the error in the RPL dialog. Note, the error message is repeated in the diagnostics output.

Diagnostics

3. Diagnostics

When intermediate information is required to debug a RPL execution error, Diagnostics can be used. Following are links to the Diagnostics section to describe how to set up these diagnostics for various sets.

- General - **HERE (Diagnostics.pdf, Section 1)**
- Rulebased Simulation - **HERE (Diagnostics.pdf, Section 3)**
- RPL Expression Slots and MRM RPL sets - **HERE (Diagnostics.pdf, Section 5)**

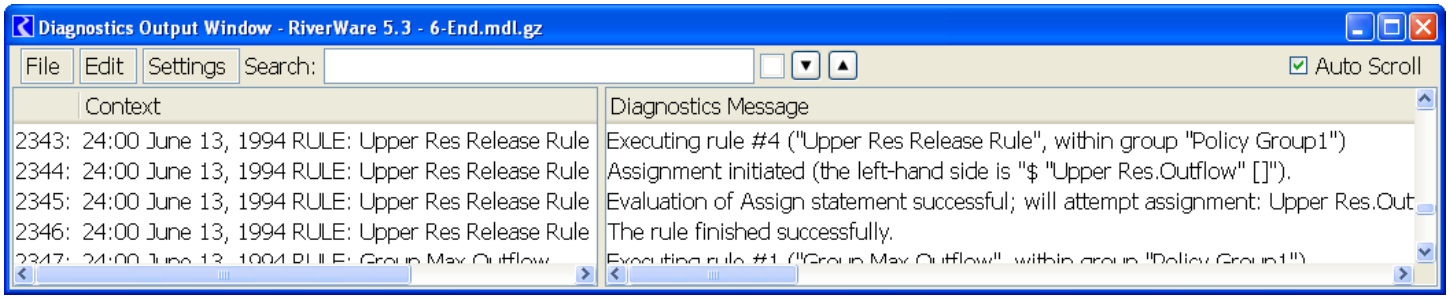
3.1 Units for RPL diagnostics

Whenever a RPL value is displayed in the diagnostics or in tooltips, RiverWare tries to use the values from the Unit Scheme, Unit Type Rule. Click **HERE (Units.pdf, Section 2)** for more information. But, for monthly and annual values, it is sometimes not known within the diagnostics for which timestep the value is associated. In that case, monthly values assume there are 31days in the month. Annual values assume 365days in the year.

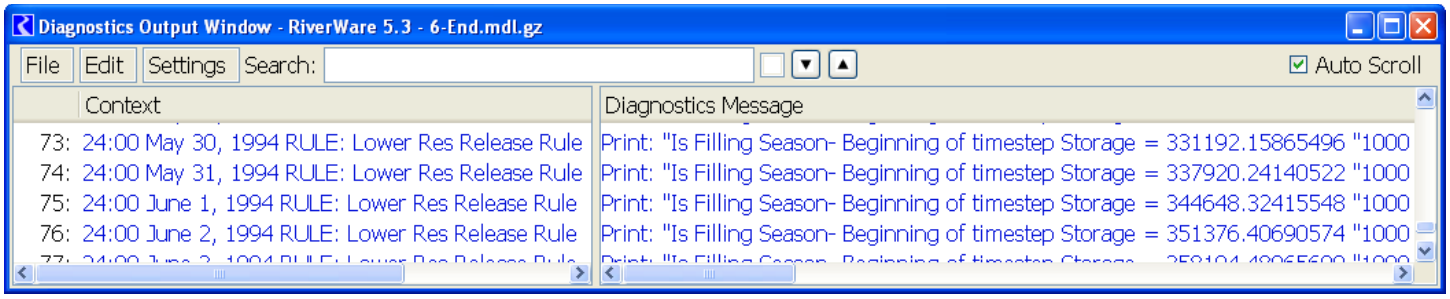
3.2 Useful RPL debugging diagnostic categories

Some diagnostics groups are more useful in debugging errors than others. The diagnostics groups are listed below, ordered from the most frequently used to the least frequently used. This text was written for debugging Rulebased Simulation rulesets but can be applied to many of the other RPL sets as well.

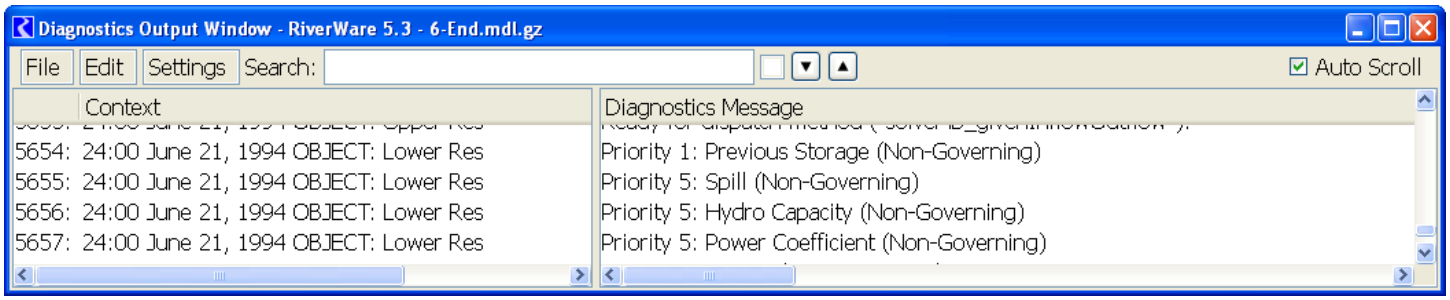
- The **Rule Execution** diagnostics group provides information about the evaluation of rules, including when they fire, whether they evaluate successfully, whether they attempt to set a slot, and the value which they attempt to set.



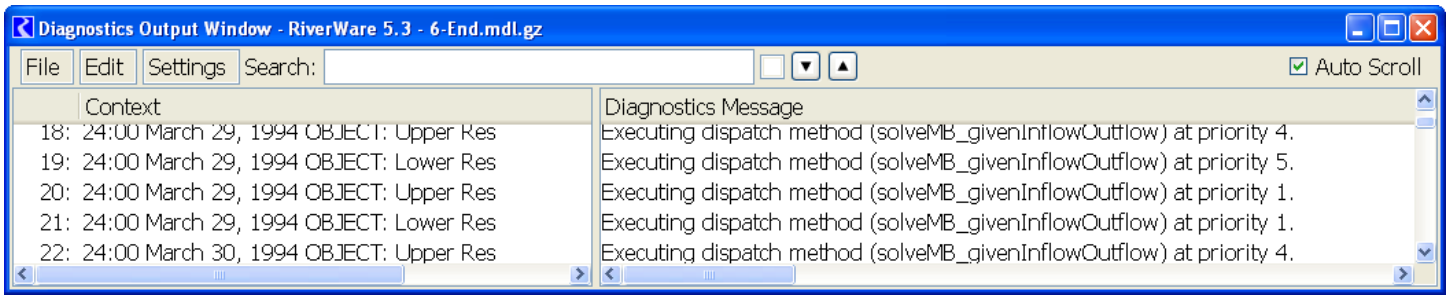
- The **Print Statements** diagnostics group enables the printing of PRINT statements in rules. If this diagnostics group is not turned on, no PRINT statements will be displayed in the Diagnostics Output Window.



- The **Dispatch Management -> SimObj** diagnostics group provides information about the priorities of slots used to determine the dispatch method for each object. This is especially critical when evaluating whether an object dispatched with the intended method.

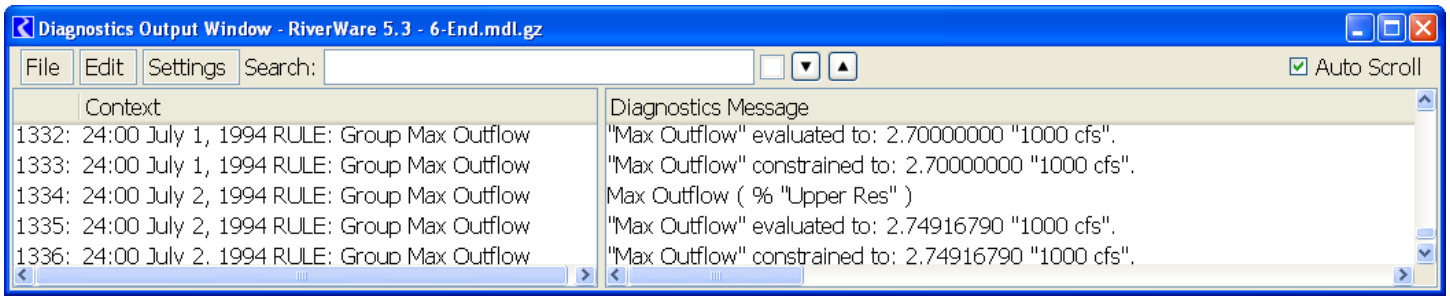


- The **Dispatch Management -> Controller** diagnostics group generates a message whenever an object begins dispatching and provides the controller priority during the dispatch.

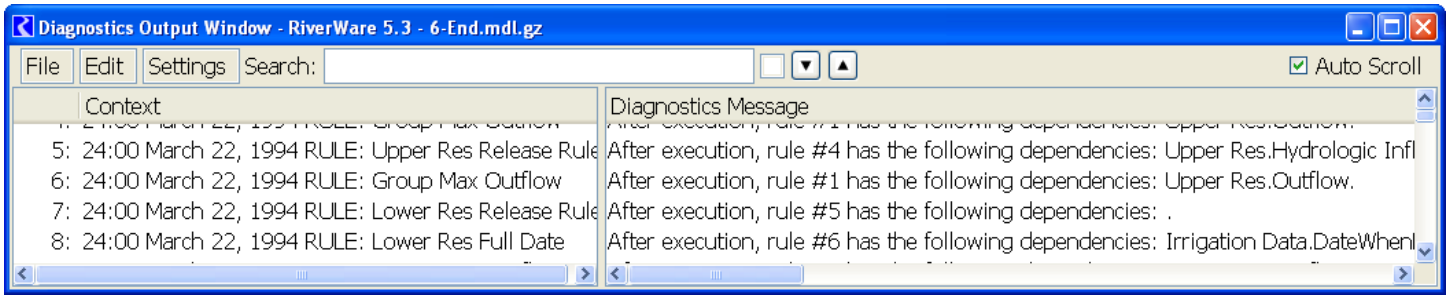


- The **Function Execution** diagnostics group provides information on the execution of predefined and user defined functions. When this group is enabled, function diagnostics are posted for all functions that have “Before Execution” and/or “After Execution” diagnostics enabled. “Before Execution” diag-

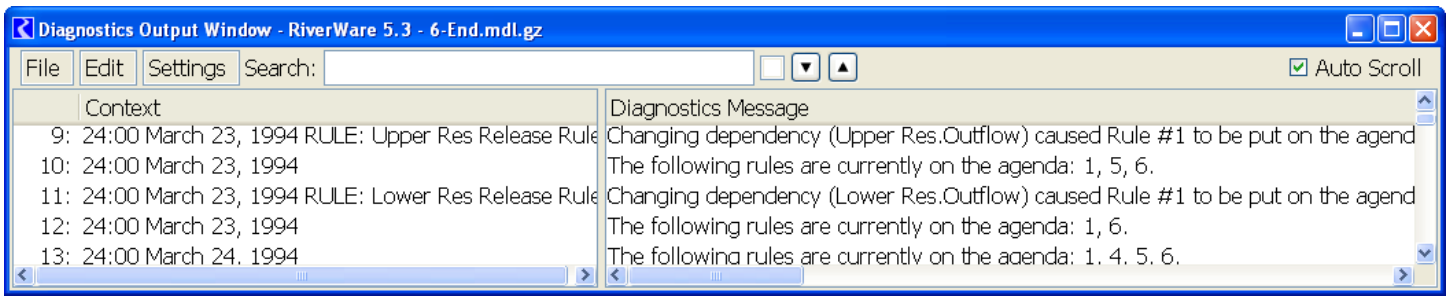
nostics show when the function is called and provides the arguments passed into the function. “After Execution” diagnostics show the result from the function call and any post-execution constraints imposed.



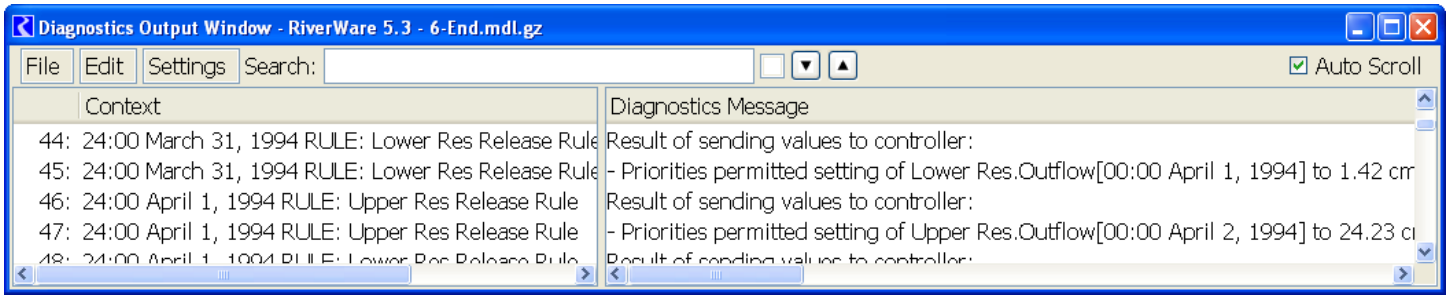
- The **Rule Management -> Dependencies** diagnostics group provides information about the dependencies of the current rule being evaluated. After the rule execution is complete (successful or not), this diagnostics group posts a list of all slots which are dependencies for this rule.



- The **Rule Management -> Agenda** diagnostics group provides information about the state of the Agenda. After each rule executes, this diagnostics group posts a list of all rules on the Agenda. The diagnostics group also posts a message whenever a rule is placed back on the Agenda due to a change of one of its dependencies.



The **Rule Management -> Cache** diagnostics group provides information about all of the slots which a rule is attempting to set. All slots are set as a group only when the rule completes successfully and all of the slot assignments are verified for priority and maximum iterations. This diagnostic group prints the result of the attempted assignments.



4. Rulebased Simulation Model Run Analysis Tool

The Rulebased Simulation Analysis Dialog is a great tool for identifying the source of fatal runtime errors. The Rulebased Simulation Analysis Dialog provides a snapshot of the last known state of the objects at each timestep. Its detail dialog shows the priorities of each dispatching slot and the last dispatch method used. This information can be used to identify priority conflicts which are the source of many errors.

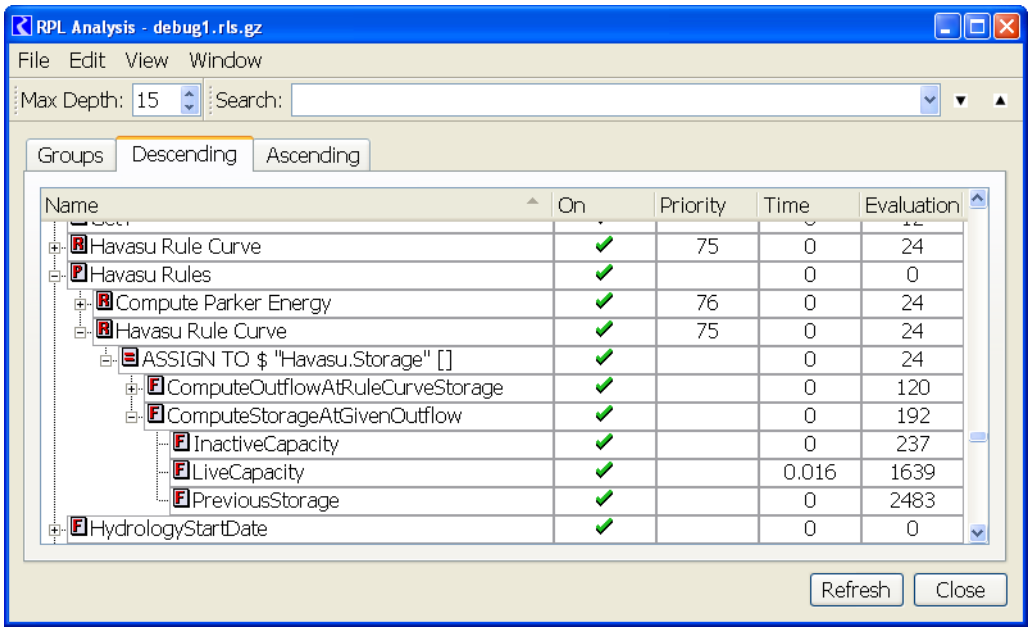
The rulebased simulation **Model Run Analysis** tool is described **HERE (ModelRunAnalysis.pdf, Section 2)**

RPL Analysis Tool

5. RPL Analysis Tool

5.1 Purpose

The RPL analysis tool is intended to assist the user in analyzing and documenting a RPL set. For this section, "items" refer to rules, groups, assignments, statements, predefined functions and user defined functions. In particular, this dialog displays two general types of information on items in the RPL set:



- Relationships amongst items within a RPL set, i.e. a listing of what functions are called by a rule or block or what rules or blocks call a given function.
- Performance information on each item in the RPL set. In addition, name, state, description and arguments are listed.

This information is useful for a number of common user tasks.

Performance: Examining the calling relationships and performance data of rules, blocks, and functions can be useful when analyzing and improving the performance of a RPL set.

Analysis and Understanding: Navigating through the calling-tree while reading the description text can be useful for interpreting a RPL set.

Documentation: Printing a portion of a calling-tree along with its item's descriptions and state provides concise documentation of a RPL set.

5.2 Overview of the RPL Analysis Dialog

The RPL Analysis Dialog is available for each of the RPL sets in RiverWare including rulesets, object level accounting method sets, expression slot RPL sets, iterative MRM rulesets, and RPL based optimization sets. In this document, a *block* refers to the upper level expression for each of these sets and will be used in all descriptions. A block is analogous to such items as a rule, method, or expression slot assignment.

5.2.1 Display of items and Data

The RPL analysis dialog allows the user to examine a wide range of information about any of the items in the RPL set. The user can customize the information that is displayed for all the items in the dialog. Among the list of information to possibly display (from the **Window ➥ Columns** menu) is: name, description, active, return-type, num arguments, argument list, item-type, priority, out-degree, in-degree, time, evaluation, and orphan.

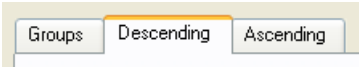
In addition to the set, groups, functions, and blocks, the dialog provides access to other items. This is useful for examining performance information and relationships amongst predefined functions, blocks, and functions.

The dialog also displays RPL statement items inside rules or blocks, such as assignments, for-each statements, and print statements. These statements are assigned names based on the left-hand-side of the statement, such as “ASSIGN TO Reservoir.Inflow[]”.

The complete list of items which can be displayed or filtered (from the **Window ➥ Objects** menu) is: sets, groups, blocks (rules), print statements, statements, user functions, and predefined functions.

5.2.2 The Views

The second major component of the RPL analysis dialog is its three expandable treeviews. These treeviews display different relationships between items in the RPL set. These are all static relationships, i.e. relationships based on the definition of the RPL set not based on any specific model run.



By default these treeviews are *linked*, so that selecting an item in one of the treeviews selects and scrolls to this object in the other two views. Thus, the three treeviews can be thought of as different views on the same focused object. The user can quickly examine the selected object from the different perspectives of these three treeviews:

- **Groups view.** The first treeview is the classic view of utility and policy groups and their members as displayed in the RPL set editor dialog. It is provided here as an entry point into a RPL set because it will be familiar to users. This view differs from the RPL set editor dialog, because it also includes the RPL Statement items defined for each block or function.
- **Descending view.** This treeview displays the items that are called by a given item. Expanding the treeview under an item reveals the items that are called by the given item. This view is called the *descending view* because it provides a *descending* view of the static call graph.

- **Ascending view.** This treeview displays the items that call a given item. Instead of descending the static call graph as in the previous view, it *ascends* the call graph to reveal all the items that call the given items.

The descending and ascending views both provide views onto the static call graph of the entire RPL set. By selecting an item in either of these views, the user can examine the entire portion of the call graph containing the selected item; the descending view shows the call graph below the item and the ascending view shows the call graph above the item.

5.3 Using the Dialog

The RPL Analysis Tool is automatically enabled for all RPL sets to view the relationship amongst items in the RPL set.

5.3.1 Opening the RPL Analysis Dialog

The RPL analysis dialog can be opened from any of the RPL editor dialogs. Each editor dialog has a menu item for "Analysis..." (i.e. depending on the type of RPL set, the menu is called **Ruleset**, **Set**, **Methods**, etc). For example, from the ruleset:

- From the ruleset editor: **Ruleset ➥ Analysis...**
- From the RPL group editor: **Group ➥ Analysis...**
- From the Rule editor: **Rule ➥ Analysis...**
- From the Function editor: **Function ➥ Analysis...**

When the RPL analysis dialog is opened from a RPL editor, the editor's item will be the currently selected item, e.g. opening the RPL analysis dialog from a function editor will force the function to be the currently selected item in the analysis dialog.

The RPL analysis dialog can also be opened using the key-command **Ctrl+Y** on Windows.

5.3.2 Switching between views

As the three listviews provide different views on the same set of RPL items, it is often helpful to switch between these different views during a session. As long as the **Window ➥ Sync Views** is active by default), the three listviews will focus on the same selected item in all three views.

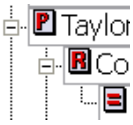
The user can switch between views by selecting the desired tab: Groups, Descending, or Ascending. In addition, the View menu provides menu items and key-commands for rapidly switching between views: **View ➥ Groups View** or **Ctrl+G**, **View ➥ Descending View** or **Ctrl+D**, **View ➥ Ascending View** or **Ctrl+A**.

5.3.3 Navigating within a treeview

The listviews provide several ways of navigating up and down.

- Use the mouse to move the vertical scrollbars up and down.

- Navigate up and down using the keyboard using the **Up-arrow** and **Down-arrow** keys.
- Pressing a letter on the keyboard will navigate to the next item that begins with the given letter.
- The user can move between children and parents in several ways.
- The [+] and [-] buttons on the left side of each row to expand and collapse the row using a mouse click.



- The **Left-arrow** and **Right-arrow** keys on the keyboard expand and collapse rows and navigate between children and parents. The first arrow press moves up or down a level and the second arrow press expands or collapses the level.
- **View ➥ Expand All** or **Alt + Right-arrow** will expand all the rows in the current listview. Note that for very large RPL sets, expanding all rows may require significant computational time.
- **View ➥ Collapse All** or **Alt + Left-arrow** will collapse all the rows in the current listview.
- **View ➥ Expand Children** or **Ctrl + Right-arrow** will open all child rows of the current item. This is a quick way to expand just a portion of the current treeview.
- View ➥ Collapse Parents or Ctrl + Left-arrow will close all the parent rows of the given item. This is a quick way of collapsing a large block of a treeview to simplify the current view.

5.3.4 Sorting

As with many other listviews in RiverWare, each listview can be sorted by a specific column. Clicking on the column header in either the Descending or Ascending view will sort the top-level items by that column. An arrow next to the column's name indicates the sorting column and the direction.

Sorting can be useful to quickly group together common items, such as all orphans (sort by Orphans column) or the most computationally expensive items (sort by Time). To eliminate confusion, the classic Groups view only allows sorting by priority number.

5.3.5 Search

The user can quickly navigate to an item by typing in the name in the Search toolbar. This toolbar is activated by right-clicking on the main menu bar and selecting **Search Toolbar.**

The search will stop on the next item that contains the entire search string. It is not required that the item name exactly match the search string, only that the name contains the search string.

The Search text box will preserve all the typed search text strings. These past searches can be accessed by clicking on the drop down arrow next to the text box.

A search is initiated by pressing the **Find Next** button. By pressing the arrow to the right of the button, the user can select an ascending or descending order. The button will honor the last search order.

As mentioned in the previous section (Navigating within a Treeview), pressing a letter on the keyboard will navigate to the next item that begins with the given letter.

5.3.6 Opening RPL editors

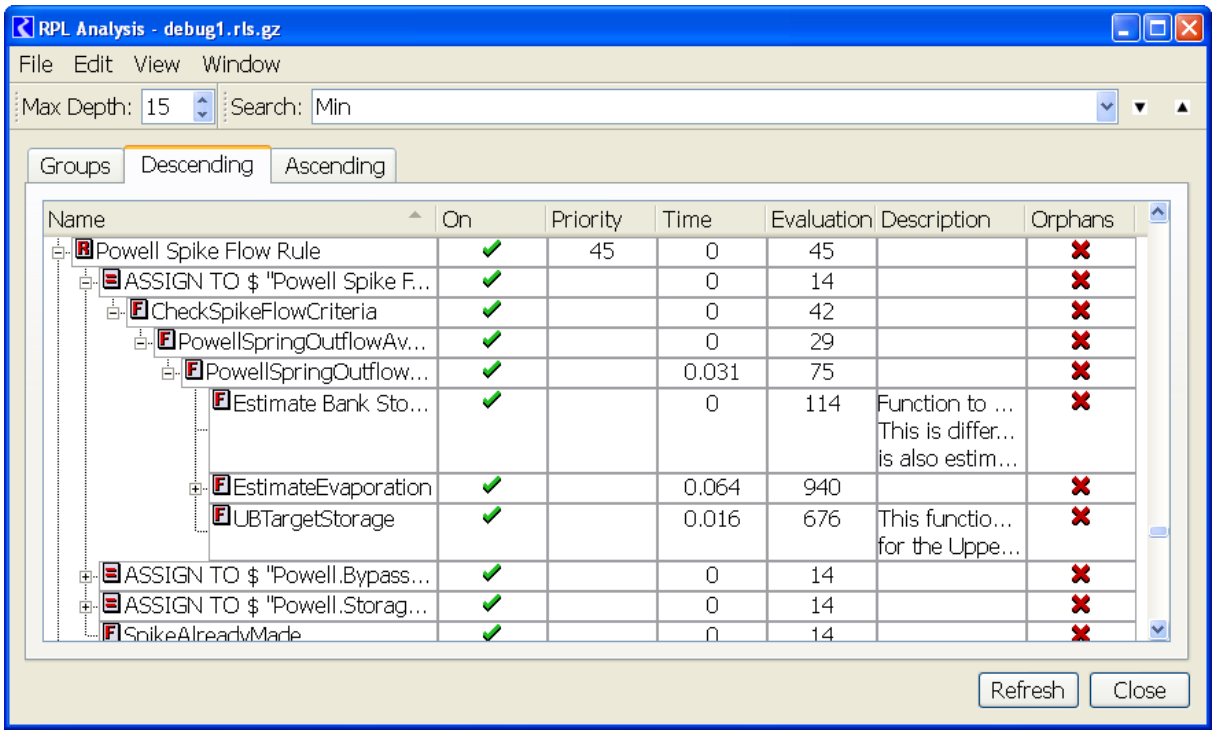
It is often useful to examine the RPL editor dialog for an item. An editor dialog can be opened several ways:

- Double-click on an item in a listview with the left-mouse button.
- Edit ➥ Open Editor or Alt+E.

For cases when the user is opening a large number of editor dialogs, the **Window ➥ Replace Existing Editor** option can be useful. With this option enabled, each new editor dialog opened will close the previous editor dialog so only a single editor is visible at one time. This eliminates the need to close each editor before opening a new one to eliminate screen clutter.

5.3.7 Customizing the views

The RPL analysis dialog has been designed to be user-customizable. With a few menu selections, the user can show only the items and information that is useful for the given task.



While the dialog can display a large amount of information, for most tasks only a subset of that information is useful. The Window menu allows the user to quickly specify which types of items and information to display in the current views.

All user settings are saved with the *user's* RiverWare preferences file. So settings made in the current session will be preserved for future RiverWare sessions run from the same login, regardless of the model or RPL set loaded.

Selecting the columns to display.

The **Window ➥ Columns** submenu contains a list of all available columns.

- **Name:** Name of the RPL item. For RPL statements, this name is generated from the left-hand-side of the statement and cannot be modified by the user, e.g. "ASSIGN TO Reservoir.Slot[]". This column also displays an icon representing the items type.
- **Description:** The user-specified description text associated with an item. This text can be edited using the item's RPL editor which can be opened by double-clicking on the item in the treeview, using the key command **Alt+E**, or selecting *Open Editor* from the right-mouse context-menu or *Edit ➥ Open Editor* from the main menu.
- **On:** Whether the item is currently active. This field can be edited from the RPL set or RPL group dialogs.
- **Return type:** The return-type of a function.
- **Num arguments:** The number of arguments that must be passed to a function.
- **Argument list:** The specific list of argument types that must be passed to a function.
- **Object-type:** The text name for the item's type. An icon representing the item's type is also displayed in the Name column.
- **Priority:** The numerical priority of a block.
- **Out-degree:** The number of RPL items called by this item, i.e. the number of children of this item in the Descending view.
- **Orphan:** An orphan is an item that is not being called by any other item. In the Ascending view, an orphan has no children.
- **In-degree:** The number of RPL items that call this item, i.e. the number of children of this item in the Ascending view.
- **Time:** The number of seconds spent in evaluating this item during the most recent model run. This value is computed using the ANSI-C clock() function which returns the number of CPU cycles the program has used (user time), measured in some system-dependent units (milliseconds on Windows). The value displayed in this column is the difference of the clock() values returned before and after executing the item, then scaled using the CLOCKS_PER_SEC macro to get a final value in seconds.
- **Evaluations:** The number of times this item was executed during the most recent model run.
- **Dispatch Count:** For a Rulebased Simulation rule, this is the number of dispatch methods that have executed due to values being set by the rule.
- **Dispatch Time:** For a Rulebased Simulation rule, this is the time spent executing dispatch methods as a result of values being set by the rule

> **Note:** For the **Time**, **Evaluations, Dispatch Count,** and **Dispatch Time** columns, the value for a RPL Set and a Policy Group is the sum of the member items. For Utility groups, no total is provided; it is always zero.

A newly activated column will be placed at the right-side of all the listviews. The order of columns can be modified by dragging the column's header at the top of the listview to a different location in the header. The same size, position, and set of columns is displayed in all three listviews.

Selecting the item types to display.

The **Window ➥ Objects** submenu contains a list of all available items. Selecting an item-type will show/hide all items of this type. Hiding a container item such as a set or group, does not hide its children.

- **Sets.** The top-level RPL set.
- **Groups.** Policy or utility groups.
- **Blocks.** block items such as rules, methods, or expression slots.
- **Print statements.** Print statements contained in blocks.
- **Other statements.** All statements, besides print statements, contained in blocks. Unlike print statements, these statements all assign values in their left-hand-side.
- **User functions.** All internal (RPL language) user functions.
- **Predefined functions**. The predefined functions and the utility groups that contain them.

5.3.8 Printing and exporting

A major goal of the RPL analysis dialog is to support documentation of RPL sets. To assist with documentation, the user is able to print or export to a tab-delimited file for importing into a third-party tool such as Excel.

Both printing and export generate output for the currently visible treeview. The user can generate output of the entire treeview or just the selected portion of the treeview. In both cases, the output will contain only the expanded rows in the treeview. If a row is not expanded in the treeview, its children will not be expanded in the output. In both cases, only the currently selected columns will be included in the output.

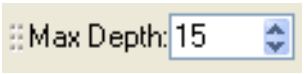
File ➥ Print ➥ All and **File ➥ Export ➥ All** generates an output of all the expanded rows in the currently visible treeview.

File ➥ Print ➥ Selected and **File ➥ Export ➥ Selected** generates an output of the currently selected rows in the currently visible treeview. Rows can be selected by: dragging a set of rows using the left-mouse button, adding a set of items by holding down the Shift key, or adding or removing specific rows by holding down the Ctrl key.

5.3.9 Customizing other behavior

In addition to the customizations for columns and item types, several other behaviors can be customized by the user. As with all customizations, these settings are saved as part of the user's RiverWare preferences which are recalled with each new RiverWare session.

- **Window ➥ Sync Views**. When this is active, selecting an item in one of the listviews will also select it in the other two listviews.
- **Window ➥ Replace Existing Editor**. When this is active, opening a RPL editor (by double-clicking on an item or using the *Edit ➥Open Editor* menu item) will close the previous RPL editor and replace it with the new editor. This saves the user from having to close each editor dialog when quickly examining the contents of numerous editors.
- **Window ➥ Show Multiple Lines**. When this is active, the treeviews will display all the lines of a multiple line description. This can be turned off when the user would rather see just the top-line of multiple line descriptions.
- **[Depth Toolbar] ➥ Maximum Depth**. Sets the maximum depth of in the treeviews. This value is typically only of interest with RPL sets that use recursion, i.e. where two functions call each other. With recursive RPL sets, the *Expand Children* or *Expand All* features need to specify a maximum depth since the recursive calls would continue infinitely. In this case the treeview will only expand to this depth.

  This toolbar can be enabled by right-clicking on the menu bar and selecting *Depth Toolbar.*