RPL Debugging and Rule Diagnostics

July 2022

Center for Advanced Decision Support
for Water and Environmental Systems
UNIVERSITY OF COLORADO **BOULDER**

Contents

1 Debugging a Rule

1.1 Introduction and Preparation

In this tutorial you will use the RPL Debugger to view the values of expressions and functions in a rule.

The tutorial is part of the RPL Debugger and Diagnostics online tutorial. See the course player for introductory and background information.

❑ Open the Arbor Basin model provided:

~\RiverWareTutorial\Models\ArborBasin_RPL_Debug.mdl.gz

The RBS ruleset is saved in the model.

❑ Run the model.

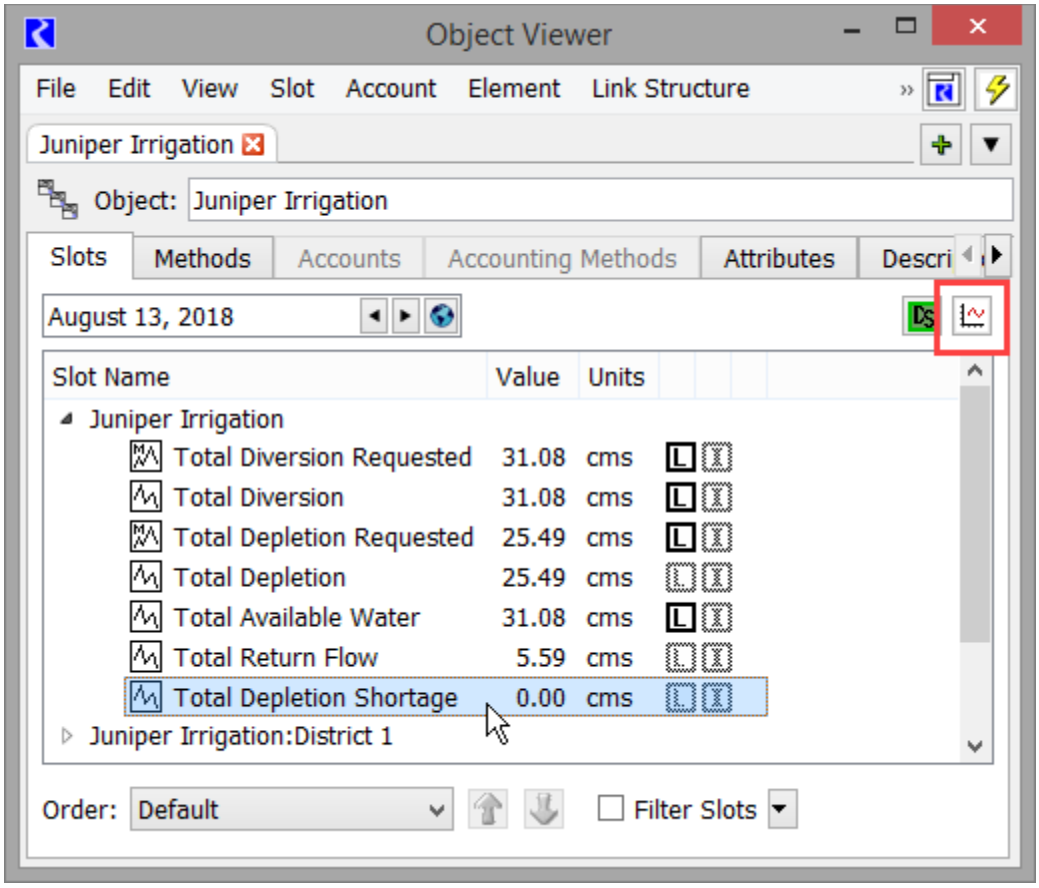
It completes successfully.

1.2 Using the RPL Debugger

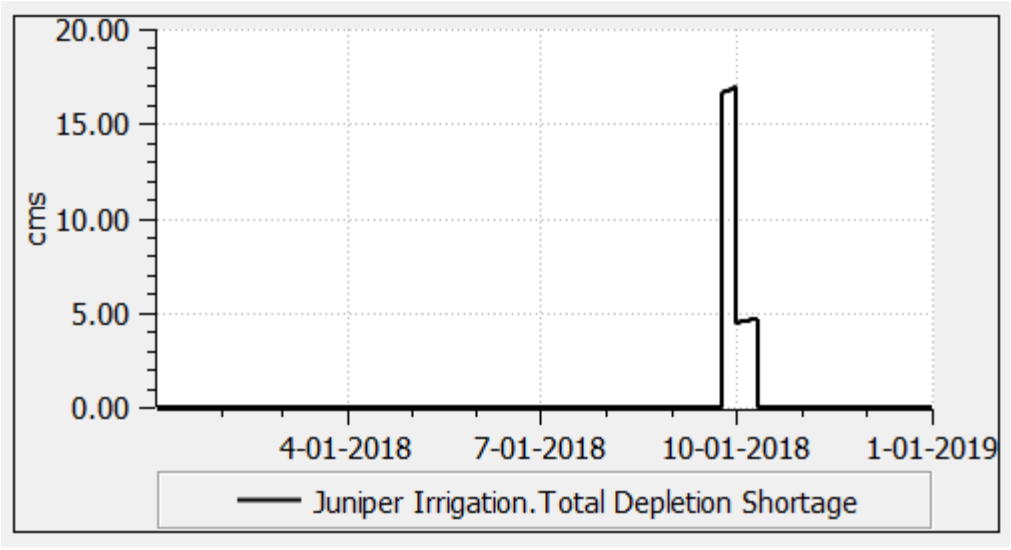
To demonstrate the RPL debugger you will investigate a diversion shortage at the Juniper Irrigation district.

❑ Open the Juniper Irrigation object

❑ Expand the first item for Juniper Irrigation.



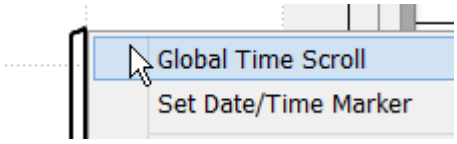
❑ Highlight the Total Depletion Shortage and click the Plot button highlighted above.



The plot shows most values are zero, meaning no shortage, except during the end of September and early October 2018. This exercise will show you how to use the RPL debugger to understand this shortage.

Let's first identify a date to investigate.

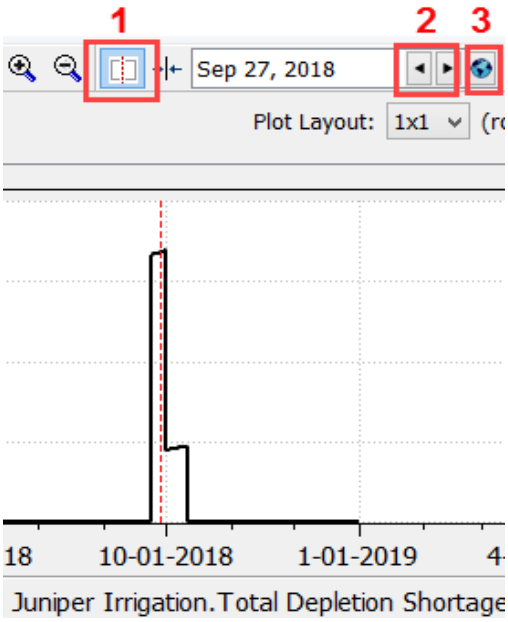
❑ Right-click on the peak of the curve and select Global Time Scroll.



Depending on where you clicked, it will global time scroll to a date between Sep 25-Oct 3, 2018

❑ Click the Date Marker button (1), below, to show the red vertical line.

❑ Adjust the datetime spinner (2) to Sep 27, 2018. We could choose any date in this range, this date is arbitrary at this point.

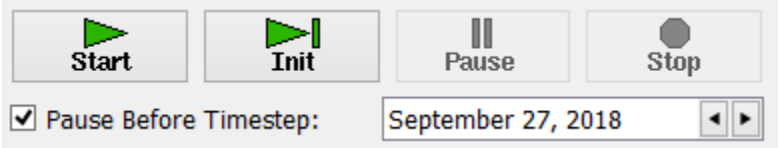


❑ Then click the Globe button (3) and select Global Time Scroll to scroll all dialogs to Sep 27th.

❑ Open the Run Control dialog box.

❑ Select the **Pause Before Timestep** check box, near the bottom.

❑ In the **Timestep** text box, enter **`Sep 27, 2018`**, or use the date selector to advance to the timestep.



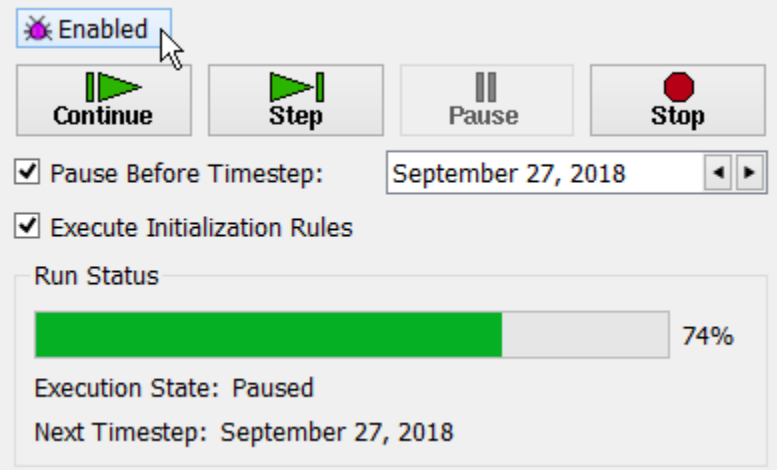
❑ Select **Start** to run the model.

The run starts and continues until September 27, 2018, where it pauses.

Before we step into the timestep, we need to do two items: enable the RPL Debugger and set a breakpoint in the RPL logic. In previous demos, we showed a number of ways to enable the debugger, we'll use an easy one here.

❑ Still on the Run Control, select **View** and then **Show RPL Debugging Button**.

❑ Click the button to enable the debugger:



Tip: Remember, enabling the RPL Debugger can slow down a model run, so it should only be enabled when necessary for debugging purposes.

Now we need to set a breakpoint in the RPL logic. To do so, we need to know the rule or function that we want to investigate. Here is a brief summary of how Hickory Outflow and Juniper diversions are set to meet Juniper demands and other objectives when water supply is limited:

- Rule 10 sets the Juniper Diversion.Minimum Diversion Bypass to pass either the gage flow or the minimum flow.
- Rule 9 sets Hickory Outflow to pass the natural flows and interstate requirements.
- Rule 8 supplements Juniper diversions from Hickory storage if the natural flows are insufficient.
- Rule 7 cuts back the Hickory diversion and releases more water to meet Juniper
- Rule 6 cuts back the Hickory diversion even further if the Hickory Pool is below minimum.
- Rule 5 cuts back Hickory outflow if the Hickory Pool is still below the minimum.

Thus, in this ruleset, we expect Hickory to release water that can then be diverted by the Juniper Irrigation District.

❑ If you aren't familiar with this model, spend a few minutes understanding the transbasin diversions from Cedar to Hickory, especially the link connections, and how water flows through to Juniper Diversions.

We want to see why Rule 8 isn't setting additional outflow for Juniper diversions.

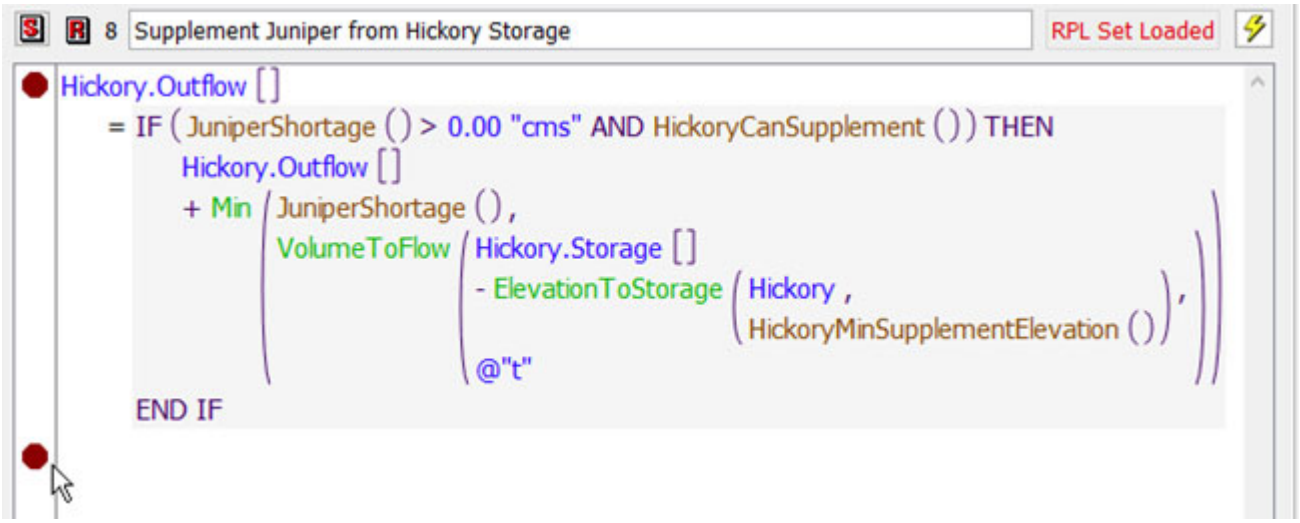
❑ Open Rule 8.

You'll notice a new column to the left of the logic. This is where you set breakpoints.

❑ Set a break point at the **beginning** of the rule by clicking at the top within the new column. See screenshot below.

❑ Set a break point at the **end** of the rule execution by selecting again in the new column.

It should look like this:



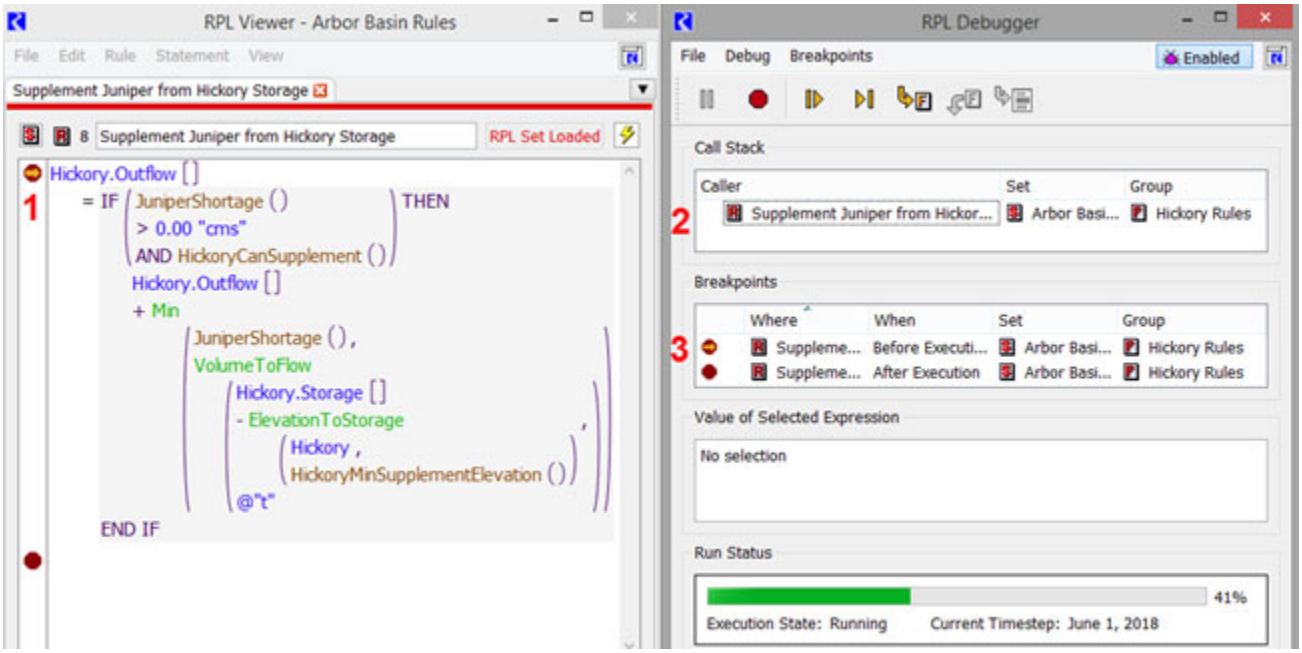
Tip: Breakpoints can go before or after statements in a rule and before after the body in a function.

❑ In the Run Control dialog box, select **Step**.

❑ The RPL Debugger will open.

The run pauses at the first break point. Notice:

1. The rule shows an orange arrow where the debugger is paused.
2. The top panel in the RPL Debugger shows the current location in the call stack.
3. The middle panel of the debugger lists the two break points.



❑ In the rule, select the function **JuniperShortage( )** to highlight it.

Notice that the bottom panel in the debugger says, "No value associated with the selection." The statement has not yet been executed, so the expressions in the rule have not been evaluated. When you see this message, it indicates the selected expression has not been evaluated either because it hasn't gotten there yet or because the logic prevents it from being evaluated, for example, the expression is part of the other clause of an IF - ELSE.

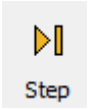
❑ In the RPL Debugger, select **Debug** and then **Show Button Labels**.

This will help you get comfortable with the buttons. Now let's step into the function and rules.

❑ In the RPL Debugger, select **Step Into** 

.

Notice the change in the Call Stack. You are now in the Juniper Shortage function, which was the first function called from the rule.

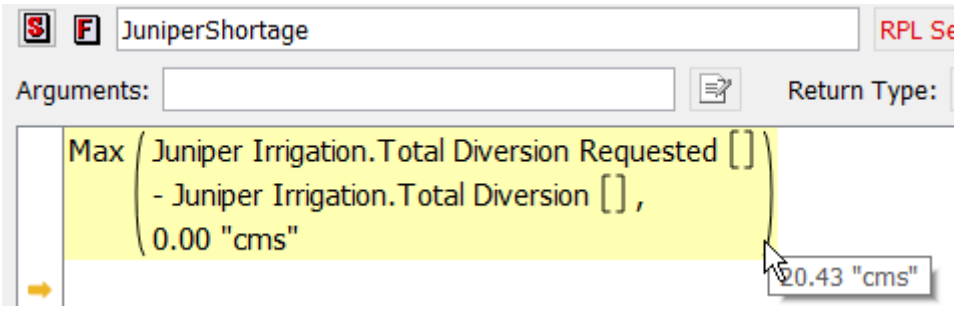
❑ Select **Step** 

in the RPL Debugger.

The orange arrow advances below the Max expression.

❑ Select the Max expression in the Juniper Shortage function.

The value is shown in the bottom panel of the RPL Debugger AND as a tool tip:



This shows that the shortage is 20.43cms.

> **Tip:** It is handy to keep the RPL Debugger and the RPL Viewer open at the same time. Line these up so you can see both as you will be using them in parallel.

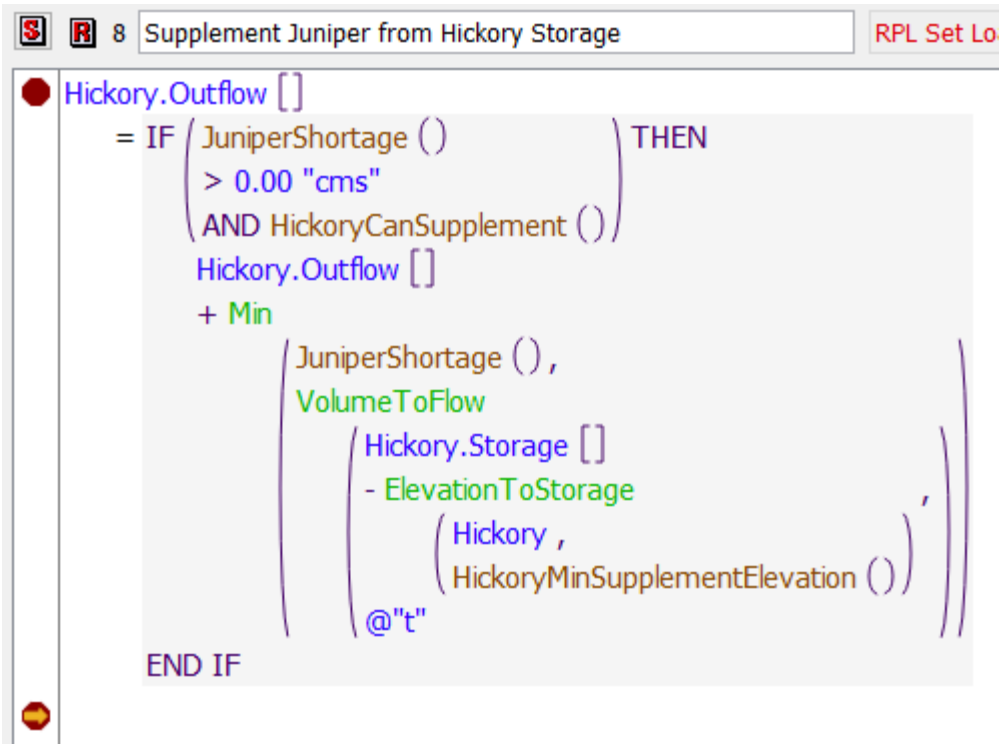
From here, we could **Step Out** to get back up to the rule and evaluate the rest of the rule logic or we could **Step** to go to the next place it could pause. In this case it would go to the Hickory-CanSupplement function.

In this and many examples, we want to fully evaluate the rule and then see which values were computed.

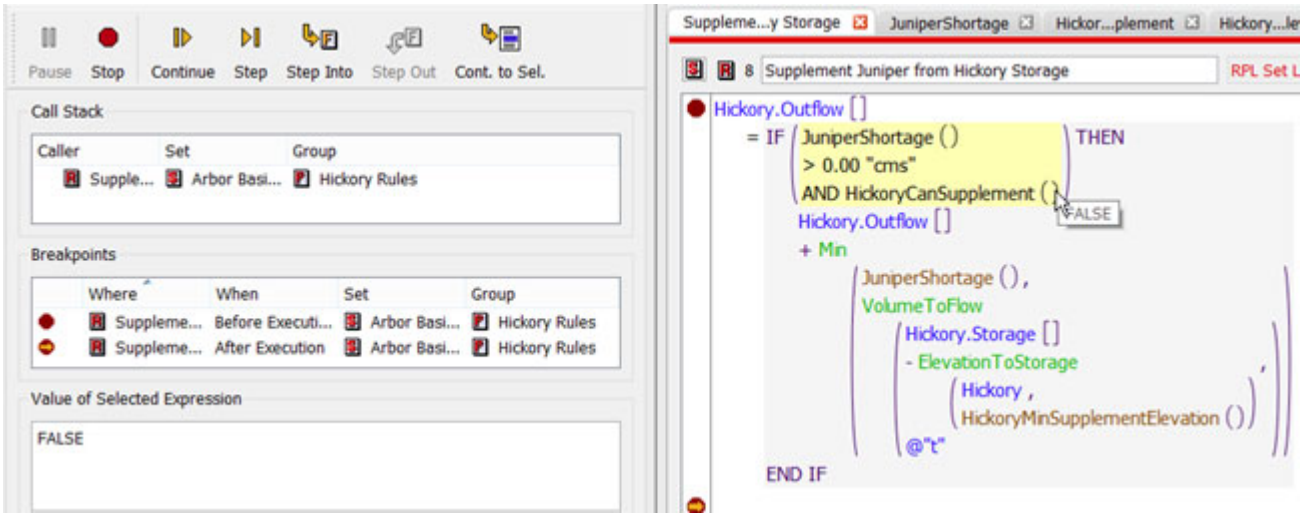
❑ Select **Continue** 

in the RPL Debugger to go to the next breakpoint.

The run pauses at the break point at the end of the Supplement Juniper from Hickory Storage rule. Now the statement has evaluated, and you can look at the values of the individual expressions.



❑ Select **AND** within the IF expression. Then look at the value displayed at the bottom of the RPL Debugger.



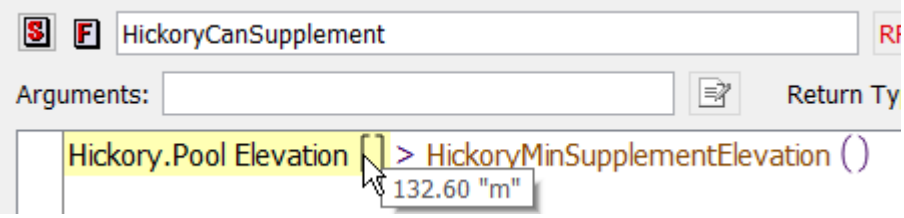
The bottom panel of the RPL Debugger and the tooltip says FALSE, indicating the boolean expression is false. We saw before that JuniperShortage was 20.4cms, therefore the Hickory-CanSupplement must be FALSE.

❑ Select **HickoryCanSupplement** and verify it returned FALSE in the debugger.

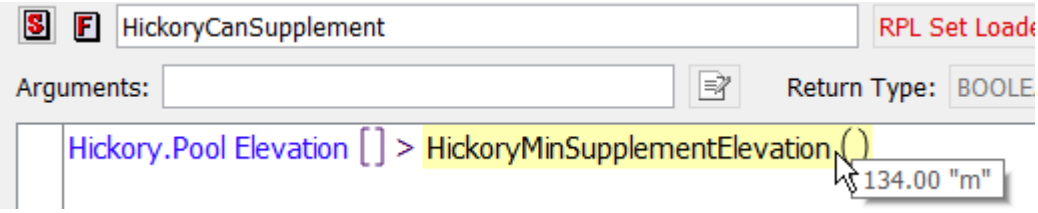
❑ Double-click **HickoryCanSupplement** to open it.

❑ Select the left expression, **Hickory.Pool Elevation[ ]**, and see that it evaluated to 132.6m

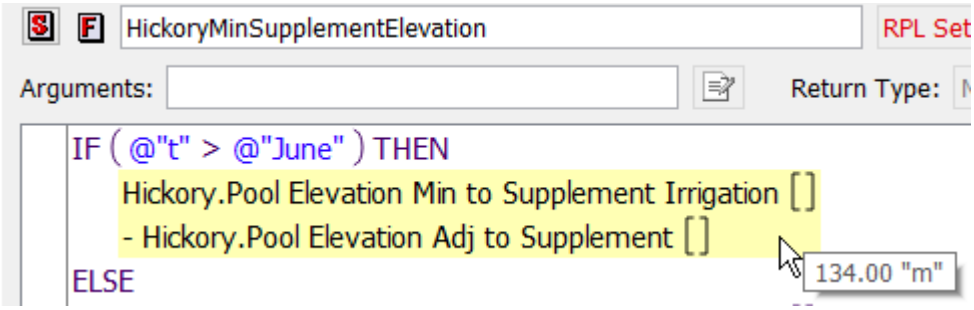
Note: In the screenshots below and throughout these tutorials, we'll show the value as a tool tip for convenience. In your use, feel free to use the Value of Selected Expression panel to see the value in the RPL Debugger.



❑ Select the **HickoryMinSupplementElevation** and see that it evaluated to 134m.



❑ Double click **HickoryMinSupplementElevation** and continue to explore the values returned.



Tip: This is the crux of the RPL Debugger, set up the breakpoints and debugger so that it pauses just after the desired statement. Then select various expressions to see their value and understand the solution.

❑ In the RPL Viewer or RPL Debugger, find the "Supplement Juniper from Hickory Storage" rule (priority 8) and show it.

Note, the debugger is paused at the rule level after the assignment statement, but you can explore the values of any expression that was computed in that logic. This is one of the most common ways to use the debugger. Set the breakpoint at the rule level, after the statement, and then drill down to find the desired values. See "4 Advanced RPL Debugging" for more advanced debugging when loops are involved.

Question: So what did we learn in this exercise?

Answer: Juniper has a shortage because Hickory.Pool Elevation is too low to supplement. The inflows are too low and no storage is available, so there is a shortage.

Let's see what happens for the rest of the timestep.

❑ First, delete the breakpoint at the start of rule 8. If click on an active breakpoint once, it will show a red outline only, indicating that it is disabled. Click on in a second time to delete it.

❑ In the ruleset, open rules 7, 6, 5, and 4 and set a breakpoint after the logic in each rule.

Now you have 5 breakpoints, one after the logic in rules 4-8. This will allow us to see the progression of execution and the values set. In Chapter "3 Using Diagnostics for Rule Debugging", we'll see how diagnostics present similar information in a single view.

Your breakpoints panel should look like this:

❑ In the debugger, click **Continue** and then look at the values computed.

Repeat to see how the timestep progresses. The following table describes the solution including the first evaluation of rule 8.

Rule	Result
8	JuniperShortage = 20.43cms; Hickory could not supplement. The rule will be ineffective
7	Hickory Diversion.Diversion Request is reduced and Hickory Outflow is increased.
8	JuniperShortage = 0.1cms, but Hickory still could not supplement
7	Hickory Diversion.Diversion Request is reduced and Hickory Outflow is increased.
8	JuniperShortage = 0.0cms so the rule will be ineffective
7	JuniperShortage = 0.0cms so the rule will be ineffective
6	The pool is less than the minimum, so Hickory Diversion is reduced.
6	The pool is less than the minimum, but there is no more Hickory Diversion to cut back. The rule will be ineffective
5	The pool is less than the minimum, so Hickory Outflow is reduced.
8	Because of reduced Hickory Outflows, there is a shortage at Juniper of 20.44cms. The pool is too low to supplement. The rule will be ineffective.

Rule	Result
7	There is a shortage and the computed diversionReduction is zero as the diversion is already zero. The rule will be ineffective.
6	The pool is equal to the Min elevation. The rule will be ineffective.
5	The pool is equal to the Min elevation. The rule will be ineffective.
4	The pool is much less than the Max elevation. The rule will be ineffective.

Now we are out of the timestep and paused in the run control.

> **Tip:** If you wanted to go through this again, follow these steps: Stop the run on the run control. Disable debugging. Start the run and let it run to September 27. Enable debugging and then Step into the timestep pausing in the RPL debugger at rule 8.

Setting a breakpoint and then pausing in each rule gives the values computed for each rule and you can often infer what values will be set on the workspace, but due to priorities the values might not get set if the slot value is already higher priority. Rule execution diagnostics provide those details on the result of the rule and would give a similar sequence of rule execution as shown in the table above. Diagnostics for debugging rules is described in "Using Diagnostics for Rule Debugging" on page 3–1.

- ❑ In the Run Control, click the button that currently shows debugging is Enabled to disable RPL debugging.
- ❑ On the Run Control, click Continue to complete the run.

1.3 Summary

In this tutorial, you saw how to use the RPL debugger to identify and highlight a value within execution of RPL within rulebased simulation.

The RPL Debugger is used to look at the individual values returned by functions and expressions in a rule at a specific timestep. This helps to verify that the rule was executing correctly.

2 Debugging a RPL Error

2.1 Introduction and Preparation

This tutorial will show you how to use the RPL Debugger when you get an error in RPL evaluation. The learning objectives are as follows:

- Learn when to turn on the debugger
- Learn how to identify the issues
- Learn how to continue out of the debugger and fix the error.

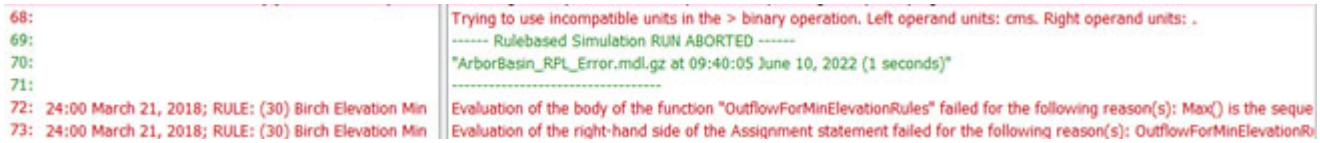
❑ Open the Arbor Basin model provided:

~\RiverWareTutorial\Models\ArborBasin_RPL_Error.mdl.gz

This is a slightly different version of the Arbor Basin model used in Chapter 1. Again, the RBS ruleset is embedded in the model.

❑ Run the model.

It errors on March 21, 2018:



The first message says what the error is and the next two red messages give the location. In a full basin model, this error could include many levels of functions calls.

The debugger, if enabled, will take you directly to the location of the error and you can explore the RPL evaluation.

This tutorial shows you how to use this feature to assist with error correction.

2.2 Enable the Debugger and Run

Let's enable the debugger let it run to the error.

❑ On the Run Control, if not already shown, select **View** and then **Show RPL Debugging Button**.

❑ Click the disabled button to enable the debugger.

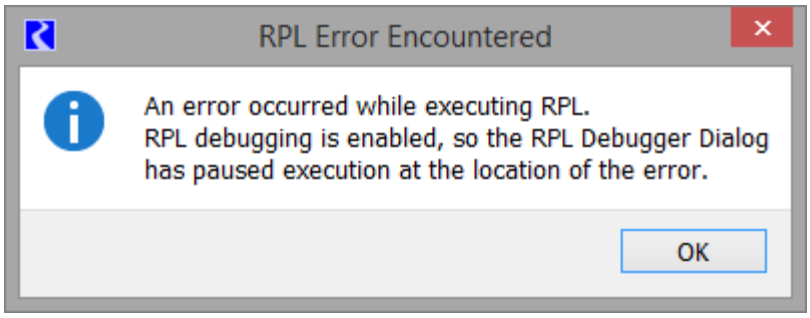
❑ Click **Start**

Because the debugger is enabled, the run progresses slowly, but is OK for this small model.

It stops on March 21 with the normal RiverWare abort notice.

❑ Click **OK**

The RPL Error notice is shown:

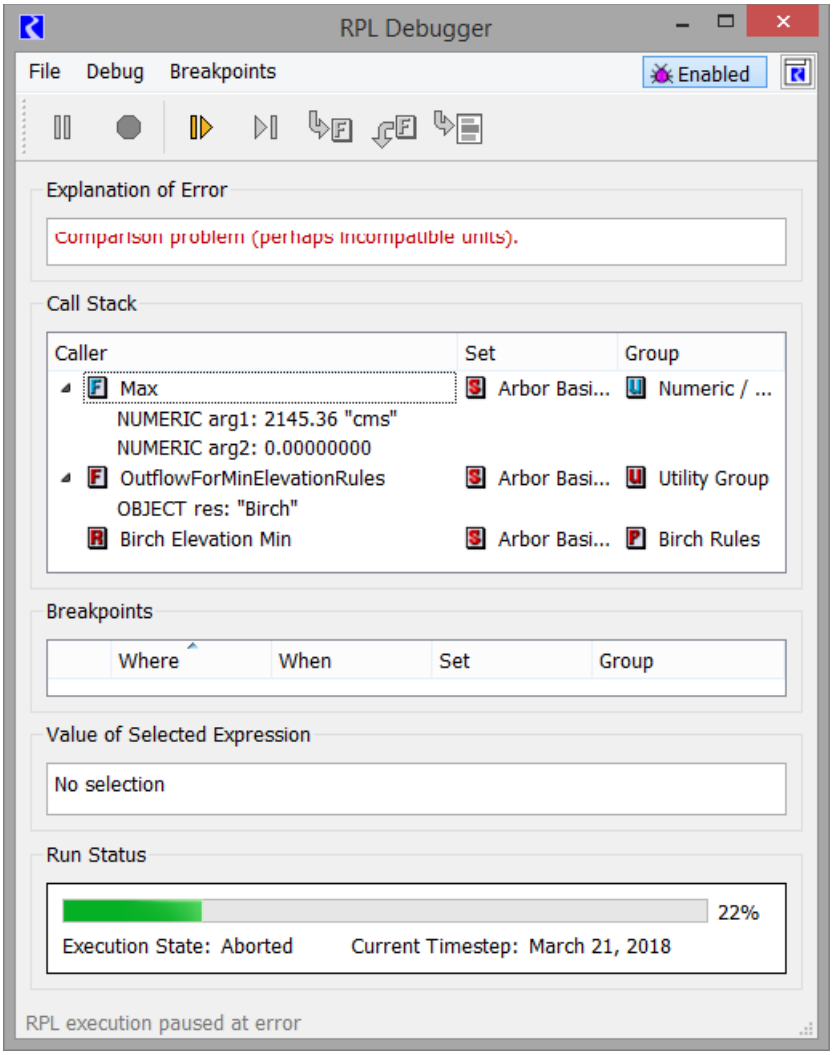


Note: Depending on the type of error and where it occurs, you may get this dialog before or after the standard Error notice.

Tip: The debugger will only open for errors in RPL evaluation. If the error occurs after a rule sets a value and the objects are dispatching, the RPL debugger will not open.

❑ Click **OK**

The debugger opens and is shown as follows. The break points panels is automatically hidden.



The Call Stack Shows the Birch Elevation Min rule called the OutflowForMinElevationRules and that called the Max function.

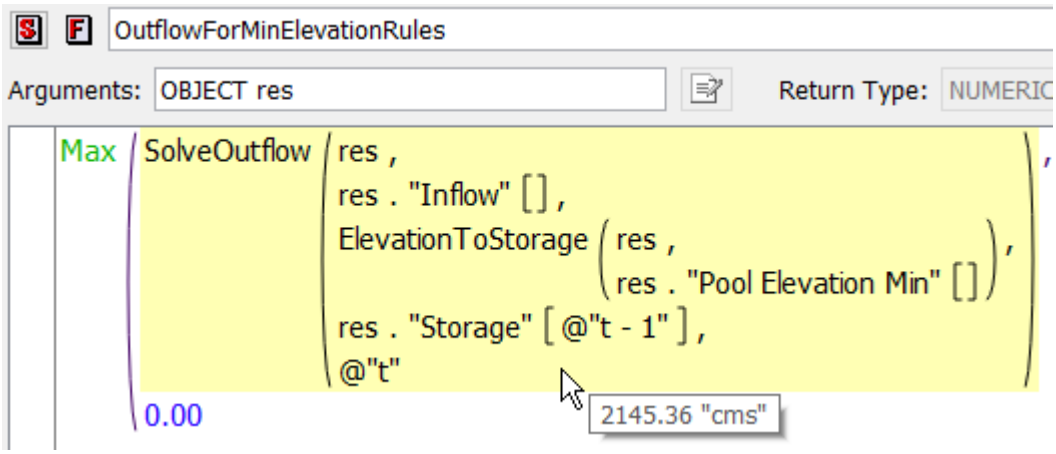
It is currently paused in the Max function

❑ Double click the **Max** function from the Call Stack.

In a predefined function, you can set breakpoints and pause, but there are no expression to select so is not as useful for looking at values.

❑ In the debugger, double click the **OutflowForMinElevationRules** function.

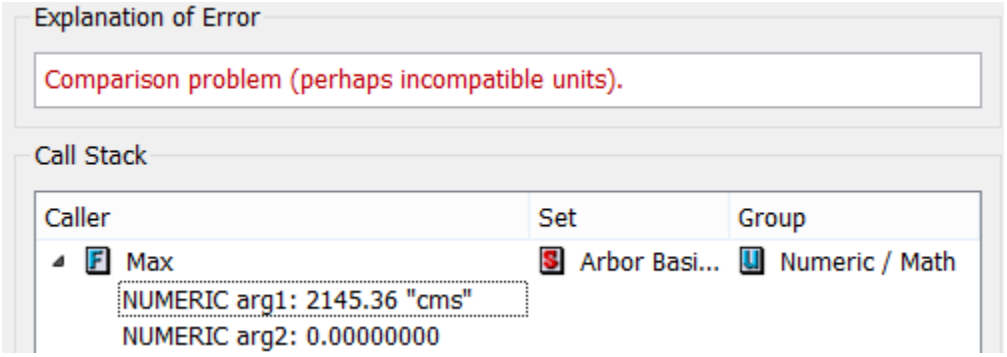
Select the **SolveOutflow** expression and notice it evaluated to 2145.36cms



The Max is comparing 2145.36cms to 0.00 with no units. That is the problem; the Max function requires unit types to be the same.

❑ Open the RPL Debugger again and in the Call Stack expand the row for the Max function.

Notice that the Call Stack in the RPL Debugger has those same values shown:



The Explanation of Error panel said it was “perhaps incompatible units”.

The Call Stack Max function showed the two arguments, one with FLOW units, the other with NO UNITS.

The fix is to add units to the 0.0 expression. To do that, you need to get out of the debugger. In this case, the stop button is disabled (we are in the middle of an abort error), so you must stop the run to make any changes.

❑ Click the Continue button

The diagnostics open and show the rest of the messages. You now have control again from the Run Control.

Find the OutflowForMinElevationRules in a RPL Viewer

❑ Double click 0.0 and change it to 0.0 cms.

❑ In the Run Control, disable the debugger and click **Start**.

It runs through with no errors.

2.3 Summary

In this tutorial, you saw how the debugger can be used to locate errors and show the values of expressions at the time the error occurred. This can make it easier to debug RPL errors as you don't have to find the problem expression and set breakpoints. The debugger does that for you.

3 Using Diagnostics for Rule Debugging

3.1 Introduction and Preparation

This tutorial shows how to use rulebased simulation diagnostics to report if a rule is executing as expected. You will learn:

- The diagnostics that are most useful.
- For an early termination, how to track down the invalid value and potential fixes

❑ Open the Arbor Basin model provided:

~\RiverWareTutorial\Models\ArborBasin_RPL_DiagnosticDemo.mdl.gz

The RBS ruleset is embedded in the model.

❑ Run the model.

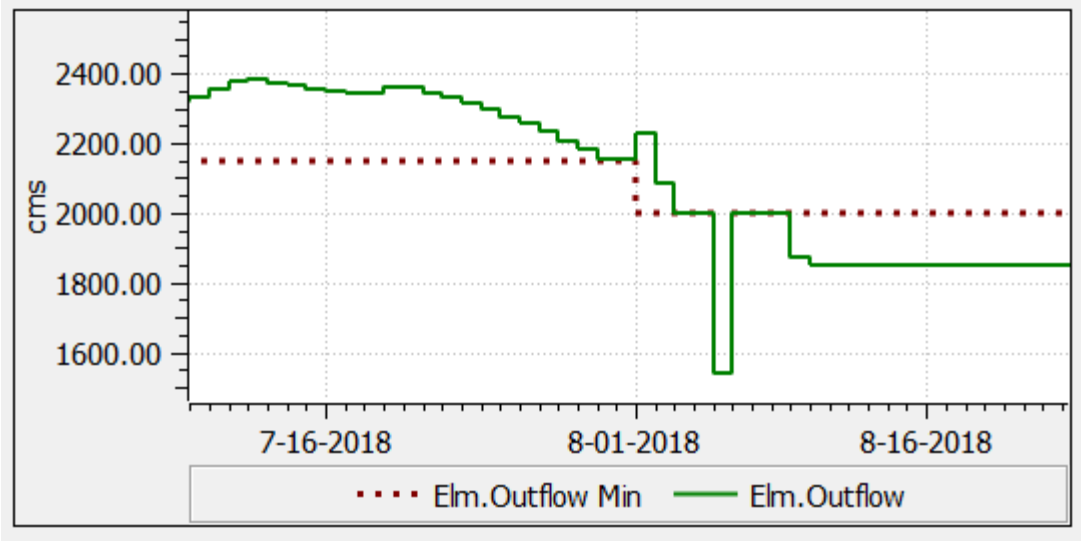
It completes successfully.

3.2 Analyzing Results

To make debugging meaningful, we need to point out a problem with the results from our Arbor Basin run.

❑ Select the **Plot** button  , then select the **Elm Outflow Minimum** plot.

The red dotted line shows the minimum flow; the green line shows the Elm.Outflow. What is happening around August 5th, 2018? Why aren't we meeting that minimum flow?



We could use the RPL debugger to try to figure this out, but in this case, it will be easier to use Diagnostics and look at the results of all rules that could set Elm.Outflow.

❑ Right click on the minimum point on the green curve on Aug 5, 2018 and select Global Time Scroll.

❑ Select **Model Run Analysis**  on the workspace main toolbar.

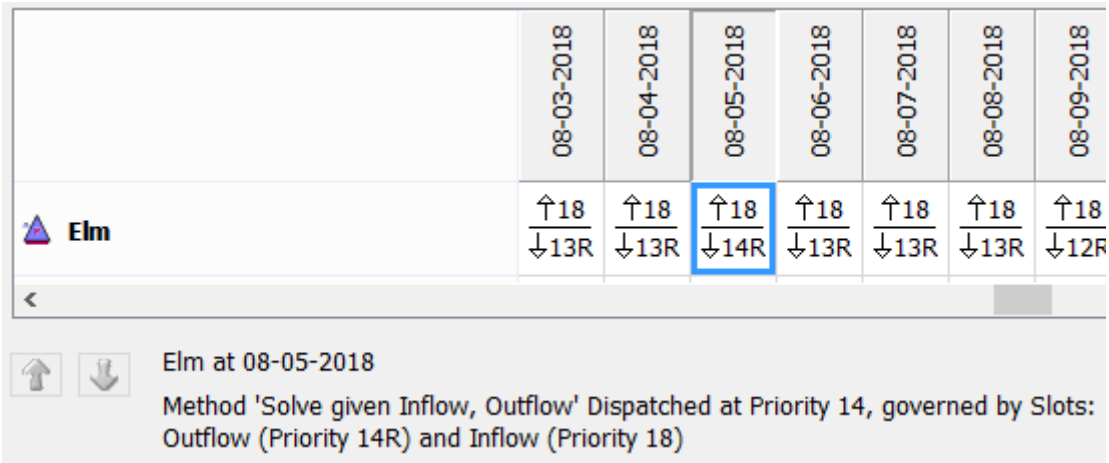
❑ In the Model Run Analysis dialog box, find Elm reservoir and it should be close to Aug 5th.

Question

Which rule set Elm Outflow on August 5, 2018?

Answer

Rule 14.



Also notice, adjacent timesteps were set by Rule 13.

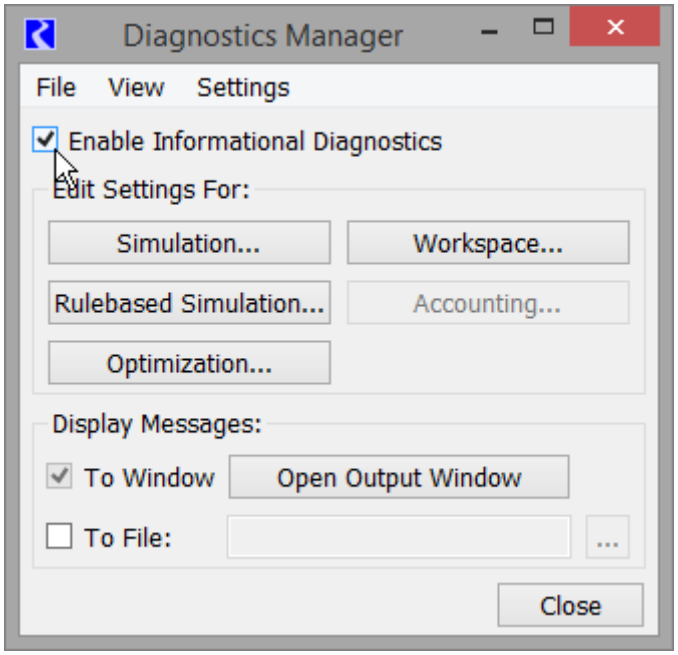
Rule 14 is the **Elm Pool Elevation Target**

Rule 13 is the **Elm Outflow Min**

This is unexpected. Rule 13 should keep the Outflow above the minimum flow requirement. You will now use diagnostics to find out why Rule 13 did not take effect on this timestep.

3.3 Using Diagnostics to Debug Rule Execution

❑ On the workspace, select **Utilities**, then **Diagnostics Manager**.



❑ Check the box to **Enable Informational Diagnostics**.

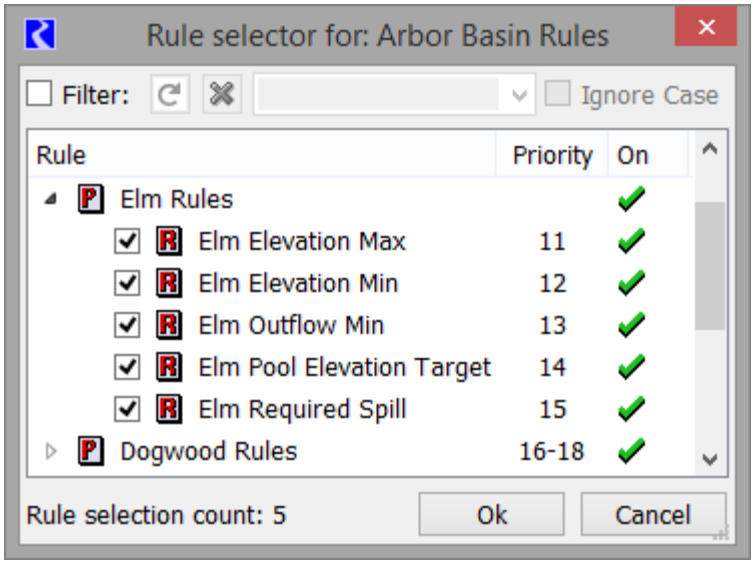
❑ Select the **Rulebased Simulation** button in the Diagnostics Manager.

❑ In the “Show diagnostics for” panel, select the **Rule Execution** diagnostics groups. See the image below.

❑ In the **Rules** context filter, click **Select**, then **RBS Ruleset**.

❑ Navigate through the rule selector and select rules 11 through 15.

Even though we know rules 13 and 14 are likely involved, we’ll print out all Elm rules for this timestep to see their results.

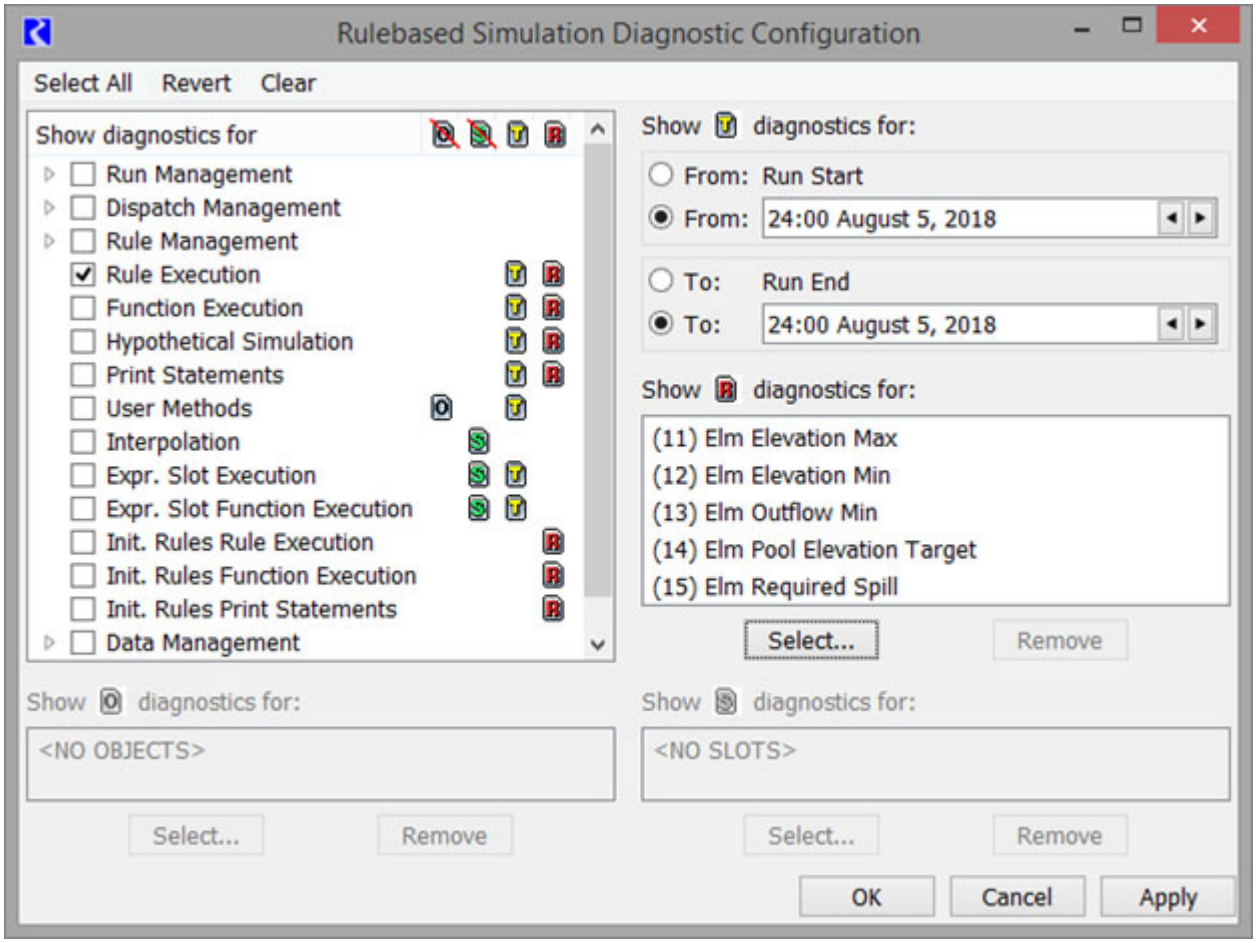


❑ Click **OK**

❑ In the **Timesteps** context filter, select the second **From:** option and enter **August 5, 2018.**

❑ Select the second **To:** option and enter **August 5, 2028** again. You only want to look at one timestep.

When you are finished, the Rulebased Simulation Diagnostic Configuration dialog box should appear similar to the following figure.



- ❑ Select **OK** in the Diagnostic Configuration.
- ❑ Select **Close** in the Diagnostics Manager.
- ❑ Rerun your model.
- ❑ From the Run Control, select **View**, then **Diagnostics Output.**

```
24:00 August 5, 2018; RULE: (15) Elm Required Spill | Executing rule #15 ("Elm Required Spill", within group "Elm Rules")
24:00 August 5, 2018; RULE: (15) Elm Required Spill | Assignment initiated (the left-hand side is "$ "Elm.Regulated Spill" []").
24:00 August 5, 2018; RULE: (15) Elm Required Spill | Evaluation of Assignment statement successful; will attempt assignment: Elm.Regulated Spill[August 5, 2018] = 400.00 cms.
24:00 August 5, 2018; RULE: (15) Elm Required Spill | The rule finished successfully.
24:00 August 5, 2018; RULE: (14) Elm Pool Elevation Ta | Executing rule #14 ("Elm Pool Elevation Target", within group "Elm Rules")
24:00 August 5, 2018; RULE: (14) Elm Pool Elevation Ta | Assignment initiated (the left-hand side is "$ "Elm.Outflow" []").
24:00 August 5, 2018; RULE: (14) Elm Pool Elevation Ta | Evaluation of Assignment statement successful; will attempt assignment: Elm.Outflow[August 5, 2018] = 1,541.00 cms.
24:00 August 5, 2018; RULE: (14) Elm Pool Elevation Ta | The rule finished successfully.
24:00 August 5, 2018; RULE: (13) Elm Outflow Min | Executing rule #13 ("Elm Outflow Min", within group "Elm Rules")
24:00 August 5, 2018; RULE: (13) Elm Outflow Min | WITH statement evaluating with res = "Elm".
24:00 August 5, 2018; RULE: (13) Elm Outflow Min | Assignment initiated (the left-hand side is ""res" . "Outflow" []").
24:00 August 5, 2018; RULE: (13) Elm Outflow Min | Evaluation of the right-hand side of the Assignment statement terminated early for the following reason: Encountered invalid value in the series
24:00 August 5, 2018; RULE: (12) Elm Elevation Min | Executing rule #12 ("Elm Elevation Min", within group "Elm Rules")
24:00 August 5, 2018; RULE: (12) Elm Elevation Min | WITH statement evaluating with res = "Elm".
24:00 August 5, 2018; RULE: (12) Elm Elevation Min | Assignment initiated (the left-hand side is ""res" . "Outflow" []").
24:00 August 5, 2018; RULE: (12) Elm Elevation Min | Evaluation of the right-hand side of the Assignment statement did not yield a result for the following reason: All conditions of an IF expression w
24:00 August 5, 2018; RULE: (11) Elm Elevation Max | Executing rule #11 ("Elm Elevation Max", within group "Elm Rules")
24:00 August 5, 2018; RULE: (11) Elm Elevation Max | WITH statement evaluating with res = "Elm".
24:00 August 5, 2018; RULE: (11) Elm Elevation Max | Assignment initiated (the left-hand side is ""res" . "Outflow" []").
24:00 August 5, 2018; RULE: (11) Elm Elevation Max | Evaluation of the right-hand side of the Assignment statement did not yield a result for the following reason: All conditions of an IF expression w
```

Messages are posted for each rule execution, starting with rule 15:

- Rule 15 sets Elm.Regulated Spill to 400cms.
- Rule 14 sets Elm.Outflow to 1,541cms.
- Rule 13's message says:

```
Elm Outflow Min: Evaluation of the right-hand side of the Assignment
   statement terminated early for the following reason: Encountered
   invalid value in the series slot "Elm.Outflow Min" at the date
   24:00 August 5, 2018. This occurred at the following location
   within the expression: "res" . "Outflow Min" [].
```

The key here is that the rule terminated early and it says the slot value that was invalid. In this case it is: Elm.Outflow Min on Aug 5, 2018. We'll follow up on this below.

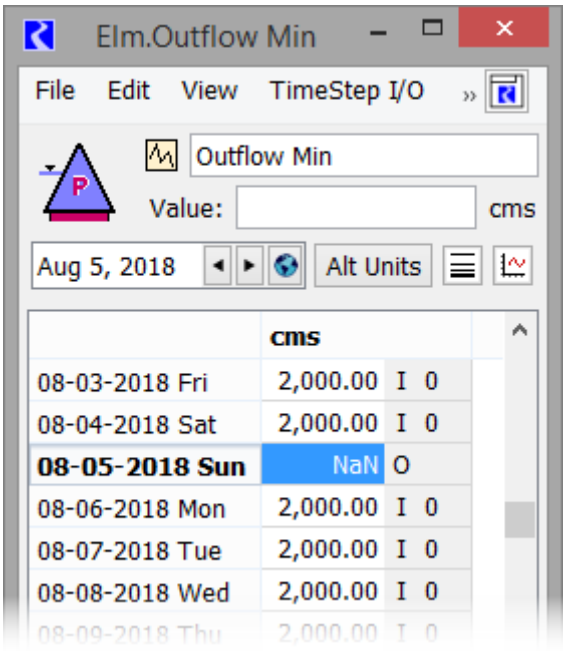
- Rule 12 says it did not yield a result because the IF was FALSE and there was no ELSE.
- Rule 11 says it did not yield a result because the IF was FALSE and there was no ELSE.

You can see how useful these messages are. They say the exact result of the rule and if it didn't succeed, it gives the slots or logic indicating the reason it didn't succeed.

Let's continue the debugging exercise and fix the error.

❑ Open the **Elm** object

❑ Find the **Outflow Min** series slot and open it.



As the diagnostic output printed, there is no value for August 5, 2018. Maybe the user missed entering this value, or it was not imported from the database by a DMI.

❑ Type 2000 in the cell for Aug 5, 2018.

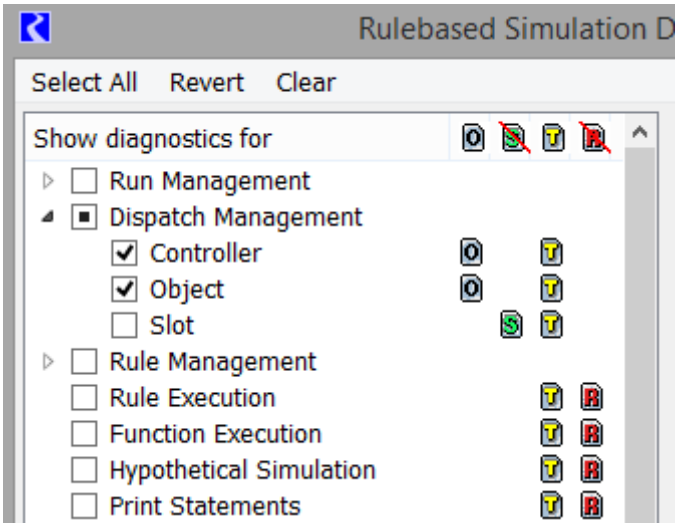
❑ Run the model

❑ Verify in the plot and the diagnostics that now the Elm.Outflow is set to 2,000 cms on 8/5/2018.

3.4 Other Useful Diagnostic Categories

When debugging rules and RPL, the rule execution diagnostics are often the most useful, but other useful diagnostic categories include:

- Dispatch Management → Controller
- Dispatch Management → Object



These two are useful if you want to know if and when an object solves after rules have set values.

> **Note:** Two other diagnostic categories, Function Execution and Print Statements, were useful before the RPL Debugger was developed. These two can be used to get detailed information when using functions and print statements. Now the RPL Debugger is preferred for debugging.

3.5 Summary

In this tutorial, you saw how to use diagnostics to debug a rulebased simulation. In particular, Rule Execution diagnostics show the result of the specified rules including the values set if successful, missing or invalid values when terminating early, and the reason for a rule finishing ineffectively.

Together with the RPL Debugger, diagnostics provide the tools necessary to debug complex logic.

4 Advanced RPL Debugging

4.1 Introduction and Preparation

This tutorial presents techniques for debugging more complex rules. In this tutorial you will learn the following:

- How units are represented in the RPL debugger and how to change the display units.
- How to debug a rule with multiple statements.
- How to view LIST expressions in the RPL debugger.
- How to debug a function that calls a FOR expression.

The tutorial is part of the RPL Debugger and Diagnostics online tutorial. See the presentation player for introductory and background information.

❑ Open the Arbor Basin model provided:

❑ **~\RiverWareTutorial\Models\ArborBasin_RPL_AdvancedDebug.mdl.gz**

The RBS ruleset is saved in the model.

❑ Run the model.

It completes successfully.

4.2 Units in the RPL Debugger

In this section, we'll show you how to change the display of units in the RPL debugger. This is useful if you need more precision or want to see the values in some other display units. Let's run to a particular point and look at an expression.

❑ On the Run Control, select the **Pause Before Timestep** option and make sure it says November 28, 2018.

❑ Run the model by selecting **Start**.

❑ Once paused before November 28, enable the RPL debugger (**View** and then S**how RPL debugging button**, click the button to enable).

❑ On the Run Control, **Step** into November 28.

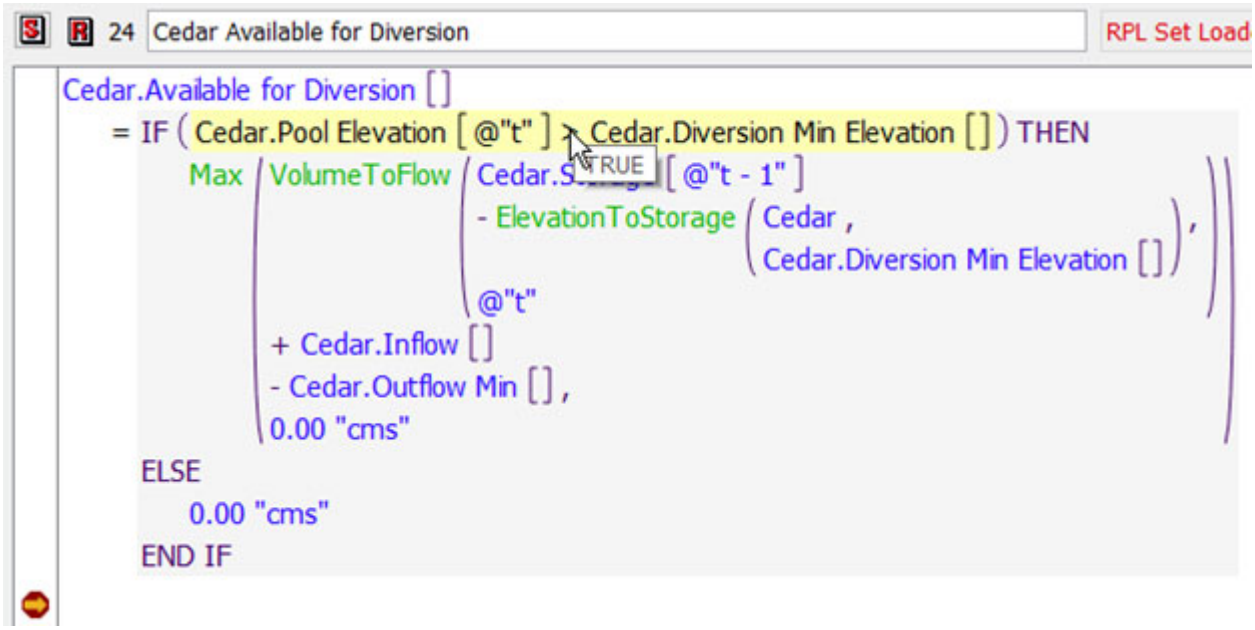
The debugger pauses at a break point already configured at the end of Rule 24.

If you click on most expressions, it will say there is no value; the reservoir hasn't solved so the rule will terminate early.

❑ In the debugger, select the **Continue** button.

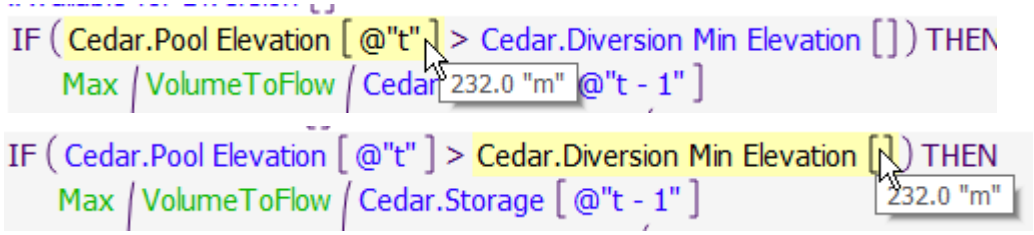
It pauses again in rule 24. Now the reservoir has solved and this rule has re-executed. Values are now shown.

❑ Select the "greater than" expression:



The value evaluated to TRUE.

❑ Select each of the expressions on either side of the "greater than".



Both expressions evaluated to 232.0 "m", yet the entire comparison says that the left operand it greater than the right operand.

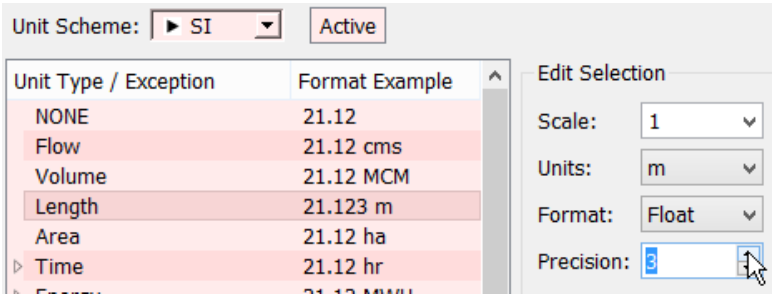
Perhaps we need to see more precision on these values because the current Pool Elevation could be just slightly above 232.0m?

The RPL Debugger uses the Unit type settings defined in the active Unit Scheme. Currently in this model, all flows are shown in cms, elevations are m, etc.

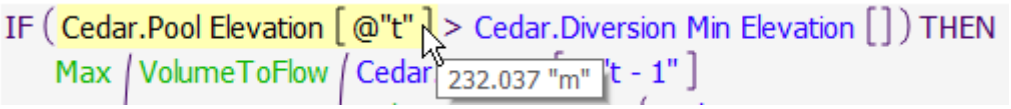
❑ On the workspace, select **Units** and then **Unit Scheme Manager.**

❑ Select the **Length** row.

❑ Change the Precision from 1 to 3.



❑ In Rule 24, reselect the expression and notice that it is now 232.037m:



❑ Verify that the Diversion Min Elevation is 232.000m.

The “greater than” is correct, Cedar Pool Elevation[@t] is larger.

This shows how you can change the display units while debugging RPL expressions. Remember that these settings include, Scale, Units, Format and Precision. You can change the Unit type settings or even activate a new unit scheme to see values with different display units.

The following are some additional items to consider regarding units in the RPL Debugger:

- The RPL Debugger always uses the Unit type setting, for example, settings for FLOW or VOLUME.
- Even if there are slot exceptions and you select that slot as the expression, the Unit type rule is used.
- For monthly and annual values, the timestep is not known within the debugger. Monthly values assume there are 31days in the month. Annual values assume 365 days in the year. It can be a little confusing if you have a monthly model and are showing flows in a per month unit, like acre-feet/month. The value shown may not look like the value that will be set on a slot if that timestep has 28, 29, or 30 days in the month.

❑ In the debugger, select **Continue** twice to get out of the debugger.

To prepare for the next section perform the following:

❑ Delete all breakpoints - in the RPL Debugger select **Breakpoints** then **Delete All Breakpoints**.

❑ On the Run Control, disable the **Pause Before Timestep**.

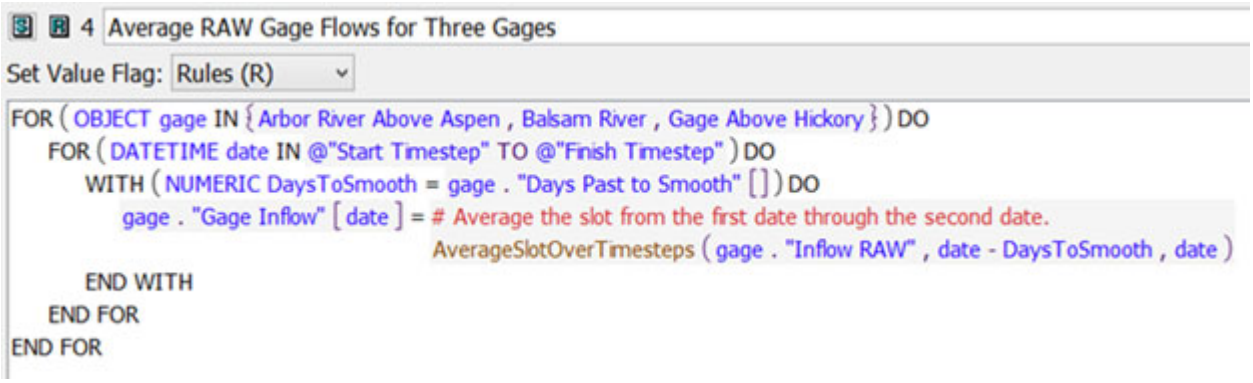
❑ Select **Continue** to complete the run.

4.3 Debugging a Rule with Multiple Statements

We will investigate two initialization rules. Remember, initialization rules are stored in a separate set that is stored in the model file.

❑ Open the Initialization set from the **Policy** and then **Initialization Rules Set** menu.

❑ Open rule 4, **Average RAW Gage Flows for Three Gages**:



This rule loops over three gages and then for each timestep in the run, sets the Gage Inflow to the average of the RAW slot over the specified number of timesteps.

❑ If not already enabled, Enable RPL Debugging as you learned in "1 Debugging a Rule"

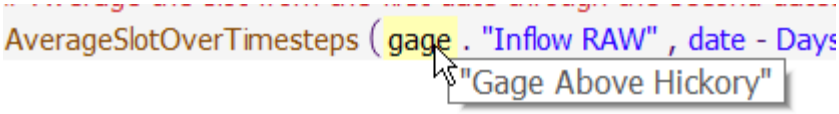
❑ Set a breakpoint at the end of the rule.

❑ On the Run Control, select **Init** to step into the initialization ruleset.

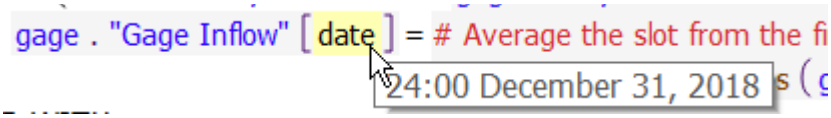
> **Tip:** The initialization rules are executed once at the very beginning of the run. As a result, when debugging or using diagnostics, you only need to **Init** to step through initialization and then you can stop the run.

The debugger is shown and paused at the breakpoint you just set.

❑ In the rule, within the AverageSlotOverTimesteps function arguments, select **gage**, and notice it says “Gage Above Hickory”.



❑ Select **date**, and notice it shows “24:00 December 31, 2018”, the finish timestep.



The rule has completely evaluated. When paused at this breakpoint, the values shown are from the last evaluation. With this breakpoint, you can’t see the values for other timesteps in the list or other gages.

How would you debug an issue on January 4, 2018 on Balsam River? To do this, we can set a breakpoint before any Statement or after the last Statement. Let’s set that up.

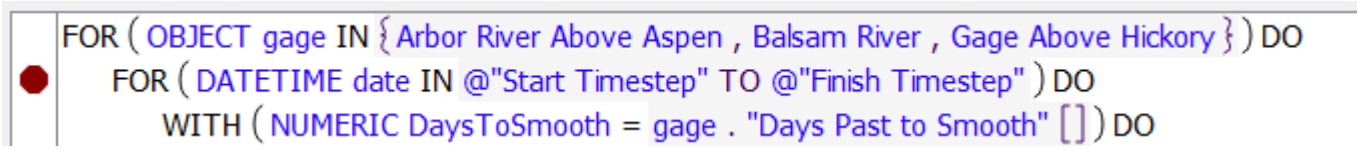
❑ First, let’s get out of this run. In the debugger, click the **Stop** button.

The diagnostics will say there was an “Error encountered,...” That just means the run was stopped.

❑ Disable or delete your breakpoint at the end of the rule.

Using Breakpoints Within For Statements

❑ Set a breakpoint on the FOR (DATETIME date... line as shown:



❑ In the Run Control, select **Init** to start the run.

It will pause at your breakpoint.

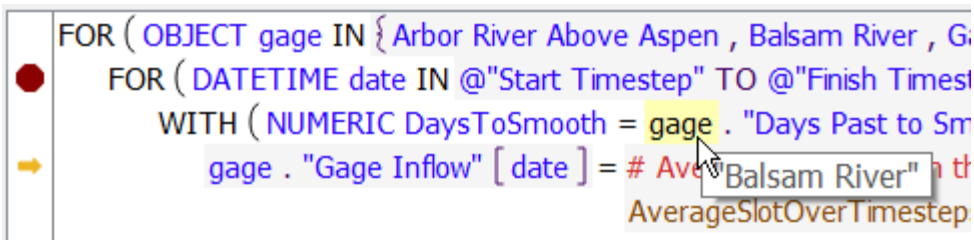
Although we can’t see it right now, the “gage” is “Arbor River Above Aspen”.

❑ In the RPL Debugger, select the **Continue** button.

Now the gage should be Balsam River.

❑ In the RPL Debugger, select **Step** twice to get to the assignment statement.

❑ Verify the gage is Balsam River.



Question: If you click the "date" on the left side of the assignment what do you think it will say in the debugger?

Answer: You might think it is the Start Timestep, Jan 1, 2018, but since we are paused before that expression, it hasn't evaluated yet for this time through the loop. Instead, it will say the last evaluation, 24:00 December 31, 2018, the finish timestep, from the loop for the first gage.

❑ In the RPL Debugger, select **Step.**

It evaluates the assignment statement and the date now shows 24:00 January 1, 2018 and shows the values computed for that date for Balsam.

In this way, it is important to remember that the values shown are from the last evaluation. When first learning how to debug, this may not be intuitive. As you practice using the debugger, this will become second nature.

❑ In the RPL Debugger, select **Step** four or five times until the "date" expression shows January 4, 2018.

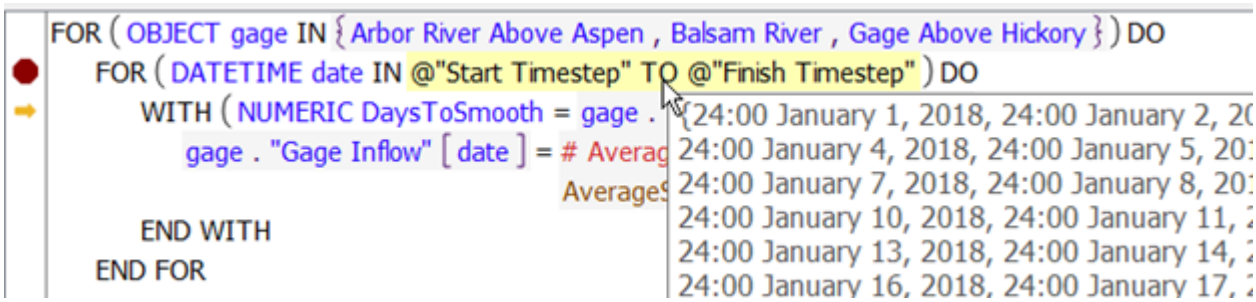
Now we can see that the AverageSlotOverTimesteps evaluates to 50.63cms and averages the Inflow RAW slot from Dec 28, 2017 to January 4, 2018. We did this by selecting each expression, for example:

We could also open the AverageSlotOverTimesteps and investigate it further.

Viewing items in a list

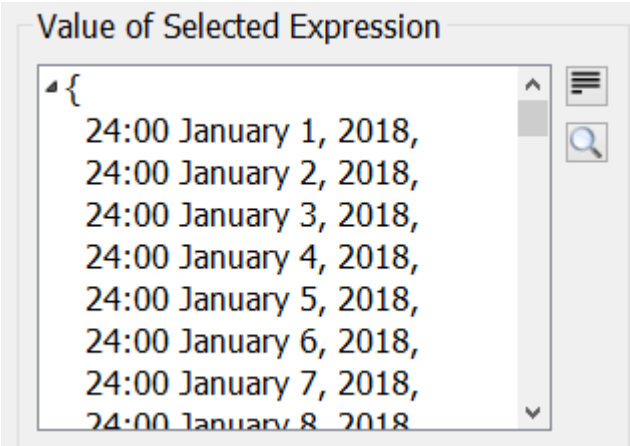
Before we get out of this run, let's look at the debugger and how it displays lists.

❑ Select the **TO** operator between the Start Timestep and Finish Timestep.



Notice the tool tip shows a long list of dates that can be hard to read.

❑ Look in the debugger and notice that this same list is presented in a nice collapsible list:



❑ Collapse the list and see it in a one-line view. Also, experiment with the buttons to the right to see the list in a flat view and use the magnifying glass button to pop out the view to a separate dialog. This can be particularly useful for very large lists.

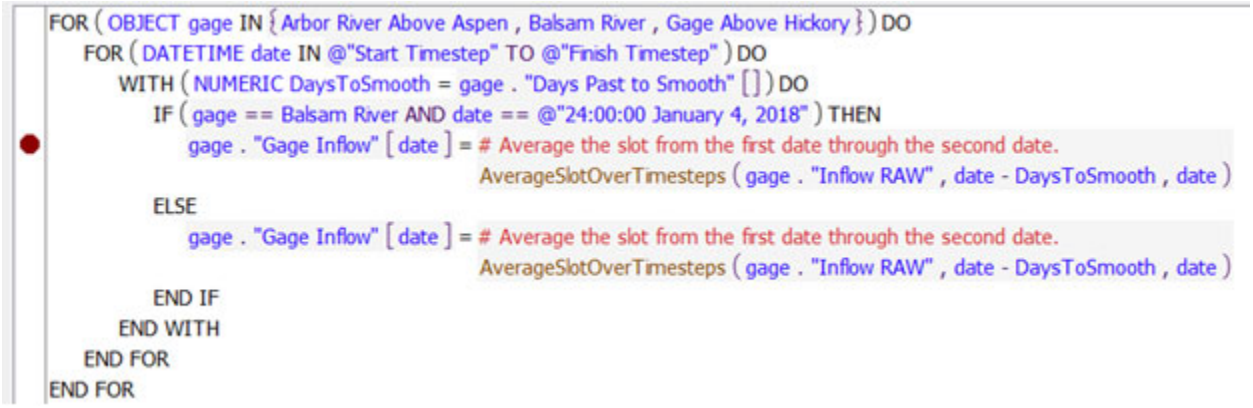
❑ From the debugger, select the **Stop** button to end/abort the run. The error message is okay.

❑ Delete all of your breakpoints from the debugger (**Breakpoints** then **Delete All Breakpoints**).

Using Temporary Logic for Debugging

In the above sections, we set a breakpoint in a statement and then used the debugger to step to the desired iteration in the loop. This worked fine for the sample problem with three gages when we were interested in the 4th timestep, but what if there were 100 gages you and needed to look at the 2,000th timestep. This would be very time consuming to step through.

Instead, consider using some temporary RPL logic to set up a place where you can place the desired breakpoint. For our sample problem of debugging Balsam Gage on January 4, 2018, we can write a new IF statement and add in a breakpoint before the assignment statement.



Note, both assignments--for the IF and ELSE clause--are the same so that the actual results do not change. The only change is to add in a place to put the debugging breakpoint.

For the screenshot above, you would then step into the initialization rule and it would pause at this breakpoint. Then Step to advance through the statement. Now you can see the exact values that were computed for that object at that timestep.

If time allows, implement the above logic and see how it works.

❑ In the debugger and/or the Run Control, select **Stop** to end the run.

❑ Delete all of your breakpoints from the debugger.

4.4 Debugging a FOR Expression

In the previous exercise, we put breakpoints within a FOR statement. Similar techniques can be used for IF and WITH statements. Remember that *statements* are only at the rule level. The other looping mechanisms in RPL are FOR, WITH, and WHILE *expressions*. What if you need to debug inside of a FOR expression? This section will provide techniques for looking at the results and debugging within FOR expressions.

❑ In the initialization rules set, open rule 3, Smooth RAW Gage Flows for Cherry.

This rule is similar to rule 4, but uses a different smoothing technique involving coefficients to adjust Cherry's RAW data into the desired inflows.

```
3 Smooth RAW Gage Flows for Cherry
Set Value Flag: Rules (R)
FOR ( OBJECT gage IN { Cherry River } ) DO
    FOR ( DATETIME date IN @"Start Timestep" TO @"Finish Timestep" ) DO
        WITH ( NUMERIC DaysToSmooth = gage . "Days Past to Smooth" [ ] ) DO
            gage . "Gage Inflow" [ date ] = # Use coefficients to smooth the flows based on the
                                            # current and previous N values.
                                            SmoothUsingCoeffs ( gage . "Inflow RAW" ,
                                                                date - DaysToSmooth ,
                                                                date ,
                                                                Cherry River.Smoothing Coeffs )
        END WITH
    END FOR
END FOR
```

❑ Double-click the **SmoothUsingCoeffs** function to open it:

```
SmoothUsingCoeffs
Arguments: SLOT slot, DATETIME FirstDate, DATETIME LastDate, SLOT coeffTable    Return Type: NUMERIC
WITH LIST DatesAndFlows = # Build up a list of flows and dates.
                          FOR ( DATETIME date IN FirstDate TO LastDate ) WITH LIST result = { } DO DO
                              result = APPEND { slot [ date ] , } ONTO result
                                              { date             }
                          END FOR
    WITH LIST ListOfFlowsSortedReverseDate = # Sort by reverse date order.
                                             SortPairsDescending ( DatesAndFlows ) DO
        # Loop through the rows of the coeffs table
        FOR ( NUMERIC counter IN GetNumbers ( 0.00 ,
                                              NumRows ( coeffTable ) - 1.00 ,
                                              1.00 ) ) WITH NUMERIC result = 0.00 "cfs" DO
            result = # sum previous amount and coeff times flow value
                     result + coeffTable [ counter , ] * ListOfFlowsSortedReverseDate < counter >
                                                 [ 0 ]
        END FOR
    END WITH
END WITH
```

This function builds a list of dates and flows and then loops over the number of coefficients, multiplying the coefficient by the flow value at a particular date and adding it to the result.

❑ With the debugger enabled (but not within a run), try to set a breakpoint within the FOR expression in the middle of the function. Notice, it only adds the breakpoint at the beginning of the function. Alternatively, you can put a breakpoint at the end of the function.

This is an important item to remember in the debugger. Breakpoints and pausing can only occur before or after functions and statements. The debugger will not break or pause within a function body.

❑ On the run control, select **Init** to step into the initialization set and pause at the beginning of the SmoothUsingCoeffs function.

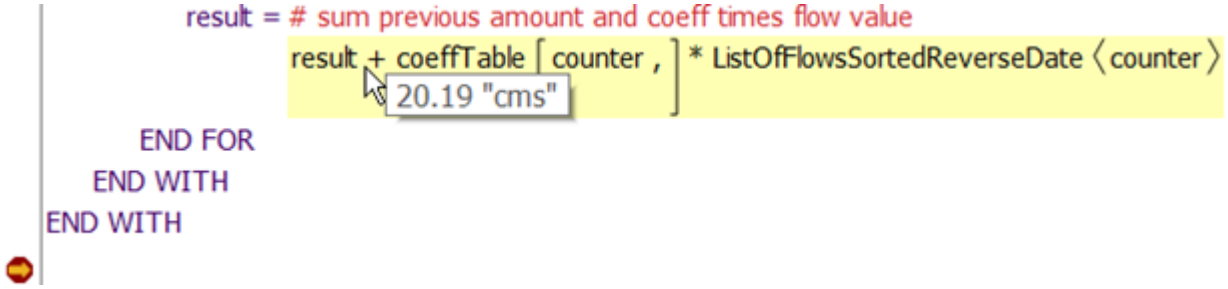
There are no values in the debugger yet as the expression has not evaluated.

❑ Set a breakpoint at the end of the function if you do not already have one.

❑ In the debugger, Step to the end of the function.

Now you can explore the results of the function.

It was called on January 1, 2018 and returned 20.18cms as shown below.



Explore other results and see how this function behaves.

❑ Continue twice and look at the next call to the function. You could keep doing this for all the dates it is called from the Initialization Rule 3.

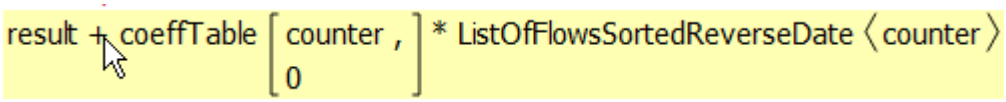
Now, let's focus on the main goal for this section, to verify the FOR expression calculation inside the function.

❑ From the debugger, select the **Stop** button to end/abort the run. The error message is okay.

❑ Delete the breakpoint at the end of the SmoothUsingCoeffs function.

To debug inside of a FOR expression, we need to call a function in which we can put a breakpoint. Since we don't have one in the SmoothUsingCoeffs function, we will need to put in a dummy or temporary debugging function. Let's edit our function.

❑ First, select the "+" expression:



❑ Copy the expression using Ctrl+C.

❑ Open the RPL Palette using Alt+P or the Function and then Palette menu.

❑ Switch to the User-Defined Functions tab.

- ❑ Double-click the **DebugFunctionNUMERIC** function.
- ❑ Select the <numeric expr> and paste using Ctrl+V. It should now look like this

```
result = # sum previous amount and coeff times flow value
        DebugFunctionNUMERIC ( result + coeffTable [ counter , ] * ListOfFlowsSortedReverseDate ⟨ counter ⟩ )
                             [ 0       ]
```

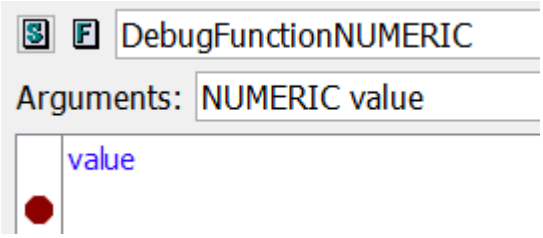
- ❑ Open the **DebugFunctionNUMERIC** and verify it returns the numeric passed into it.

  Now we can debug the SmoothUsingCoeffs function.

- ❑ On the Run Control, select **Init** to start the run.

  It will pause in the SmoothUsingCoeffs for January 1.

- ❑ Select Continue in the debugger three more times to step to January 4. (You can verify this date by clicking on **date** in rule 3).
- ❑ Add a breakpoint to the DebugFunctionNUMERIC after the body:



- ❑ In the debugger, select Continue and it should go to that breakpoint in DebugFunctionNUMERIC.

  Now we are paused in the first loop of the FOR expression.

- ❑ Go to the SmoothUsingCoeffs function and verify that **counter** is 0.0. (Indices are zero-based for table lookups)

```
result = # sum previous amount and coeff times flow value
        DebugFunctionNUMERIC ( result + coeffTable [ counter , ] * ListOfFlowsSortedReverseDate ⟨ counter ⟩ )
                             [ 0       ]  0.00000000
```

- ❑ Explore the values for the three components that make up the result: result (0.0cms), coeffTable[0,0] = 0.5, and the flow value (21.04cms).
- ❑ When satisfied, in the debugger, select continue and repeat the investigation for iteration counter 1, 2, and 3. There are four rows in the table: 0, 1, 2, and 3.

  In this way, we can use a function to debug inside of a FOR expression. In our case, we didn't have one in there already, so we added a temporary debugging function. This same process can be used for WITH and WHILE expressions.

- ❑ From the debugger, select the **Stop** button to end/abort the run. The error message is okay.

❑ If you want, in the SmoothUsingCoeffs, undo the changes to remove the debugging function. It is okay to leave it if you want, but it does clutter up the logic.

4.5 Summary

This tutorial presented techniques for debugging more complex rules. In this tutorial you performed the following:

- Learned how to debug a rule with multiple statements, especially FOR statements.
- Learned how to debug a function that calls a FOR expression.

In the process, you also learned how to look at items in a LIST in the RPL debugger, learned how to use a debug function and other debug logic, and gained additional experience using the RPL debugger. Hopefully this provides a solid foundation of techniques that you can use to debug your RPL logic.