

Technical Documentation Version 8.0

Debugging and Analysis



Center for Advanced Decision Support for Water and Environmental Systems (CADSWES)

UNIVERSITY OF COLORADO **BOULDER**

These documents are copyrighted by the Regents of the University of Colorado. No part of this document may be reproduced, stored in a retrieval system, or transmitted in any form or by any means electronic, mechanical, recording or otherwise without the prior written consent of The University of Colorado. All rights are reserved by The University of Colorado.

The University of Colorado makes no warranty of any kind with respect to the completeness or accuracy of this document. The University of Colorado may make improvements and/or changes in the product(s) and/or programs described within this document at any time and without notice.

Contents

1. Model Run Performance 1

Improving Model Run Performance 1

Performance Analysis Tools 2

RPL Analysis 2

RPL Timing and Count Predefined Functions 2

Windows Task Manager 3

Windows perfmon Utility 5

Run RiverWare More Efficiently 7

Reduce Diagnostics Processing 7

Disable Diagnostics 7

Filter Diagnostics 8

Send Diagnostics to a File 10

Break Long Runs Into Multiple Runs 10

Close Unused Windows 10

Run in Batch Mode 11

Improve Your RPL Set Logic 11

Invest in Hardware 11

Install More Physical Memory (RAM) 11

Purchase Additional, Faster Disks 12

Use Additional CPUs for Multiple Runs 13

Purchase a Bigger, Faster Machine 13

Physical Memory 13

Cache Size 13

Front-side Bus Disk Speed 14

CPU 14

Manage Memory 14

Close Other Applications 15

Adjust the Size or Location of Your Paging File 15

2. Model Run Analysis 17

Simulation 17

Model Run Analysis Dialog 19

Sorting 19

Scroll To 20

Water Quality Dispatch Information 20

Time on Top and Time on Side 20

Rules Grid 20

Show Object Icons 20

Update After Every Timestep 20

Fonts 20
Pre-run Timesteps 21
Post-run Timesteps 21
Legend 21
Dispatch State 21
Context Menus 22
Disabling Dispatching 22
Details Dialogs 22
Dispatch Behavior Details Dialogs 22
Rule Effects Details Dialog 24
Special Results Details Dialogs 24
Rulebased Simulation 25
Main Analysis Dialog 25
Dispatch Behavior Details Dialog 27
Comparing Transient and Final Information 28
Governing Rules 28
Rule Effects Details Dialog 29
3. Diagnostics 33
About RiverWare Diagnostics 33
Diagnostics Process Overview 33
Diagnostics Manager 34
Enable Diagnostics 34
Loading and Saving Diagnostic Configurations 35
Diagnostic Configuration Dialogs 35
Diagnostics Groups 36
Context Filtering 37
Select All, Revert, and Clear 39
Diagnostic Output Window 40
Message Search and Filter Controls 42
File Menu 45
Edit Menu 45
Settings Menu 46
Simulation Diagnostics 46
Rulebased Simulation Diagnostics 50
Rulebased Simulation Diagnostic Groups 51
Selecting Rules 56
Posting User Messages to Diagnostics 56
Print Statements 56
Notice, Warning, and Alert Statements 57
Stop Run Statements 57
Stop Run Expressions 57
Function Diagnostics 57
Optimization Diagnostics 59
Optimization Diagnostic Groups 59
Workspace Diagnostics 63

Accounting Diagnostics 65
Accounting Diagnostic Groups 66

4. RPL Debugging and Analysis 69

Types of RPL Debugging and Analysis 69
Building and Validation Errors 69
Errors When Building RPL Expressions 69
Errors During RPL Set Validation 70
Evaluation and Runtime Errors 72
Non-fatal RPL Evaluation Errors 72
Fatal RPL Evaluation Errors 75
Fatal Simulation Errors 77
Fatal Rulebased Simulation Errors 82
Understanding RPL Evaluation 83

RPL Debugger 83
About RPL Debugging 84
Enabling RPL Debugging 85
RPL Debugger Dialog 86
Menu Bar 88
Control Toolbar and Debug Menu 88
Call Stack Panel 89
Breakpoints Panel 90
Value of Selected Expression Panel 91
Run Status Panel 91
Using the RPL Debugger 91
Adding Breakpoints to RPL Dialogs 91
Starting the Debugger 93
Displaying Data Values 93
Data Value Units 94
Stepping, Continuing, and Pausing 94
Limitations 95
Error Handling 95
Sample Use Scenarios 96
RPL Debugger Scenario 1 96
RPL Debugger Scenario 2 97

RPL Diagnostics 98
Units for RPL Diagnostics 98
Useful RPL Debugging Diagnostic Categories 98

Rulebased Simulation Model Run Analysis Tool 101

RPL Analysis Tool 101
RPL Analysis Dialog 102
Display of Items and Data 102
The Views 103
Using the RPL Analysis Dialog 103
Opening the Dialog 103
Switching Between Views 104

Navigating Within a Treeview 104
Sorting 105
Search 105
Opening RPL Editors 105
Customizing the Views 106
Printing and Exporting 108
Customizing Other Behavior 108

Chapter 1
Model Run Performance

This section provides suggestions for improving RiverWare model run times; that is, we try to answer the question: what can you do to make your models run faster?

Topics

- Improving Model Run Performance, page 1
- Performance Analysis Tools, page 2
- Run RiverWare More Efficiently, page 7
- Invest in Hardware, page 11
- Manage Memory, page 14

Improving Model Run Performance

Following are suggestions for making your models run faster.

- Turn off unnecessary diagnostics, close as many RiverWare windows as possible, and close other applications.
- Determine if there is excessive paging during the run. If paging is a problem, purchase more physical memory. If that is not possible, adjust the paging properties.
- Analyze your model run using RiverWare utilities to determine if there are unexpected slowdowns due to excessive dispatching or slow RPL functions. Address these slowdowns by modifying your model or ruleset.
- Improve hardware; sometimes this is the most cost effective method to improve performance.
- For long production runs, use batch mode and/or divide the model into multiple runs on multiple CPUs.

  Note: If pursuing these options does not resolve the problem, there could be a modeling or internal RiverWare error. Contact `riverware-support@colorado.edu`

RiverWare attempts to perform its operations as efficiently as possible, but run times are often affected by model particulars as well as the configuration of the system on which RiverWare is run. RiverWare cannot control these factors, but many of them are under your control and these are the subjects of this section. This document attempts to be general but also attempts to prioritize the steps that will make the most difference to a user. This is very difficult to do as each user has a different system, budget, and institutional requirements. A step that may make the most sense for one user (like buying new hardware) may not even be an option other users.

Where we present information specific to Windows, examples are taken from the Windows 8.1 operating system. Although we introduce some technical concepts, e.g., RAM, CPU, and Paging, this document is not intended to be

a complete reference on computer performance issues. For more information, the reader is directed to the internet or reference books for a particular system or topic.

The remainder of this section is divided into three subsections. In the first, we describe a set of tools useful for identifying performance problems or slowdowns in RiverWare. In the second, we describe steps which may be taken within RiverWare to make models run faster. Finally, we discuss several ways to improve your computer system for optimizing RiverWare performance on your model.

Performance Analysis Tools

The first step in improving performance is typically information collection. This section describes a number of tools and approaches used to diagnose performance problems, including both tools within RiverWare as well as external utilities. The following topics are discussed:

Topics

- RPL Analysis, page 2
- RPL Timing and Count Predefined Functions, page 2
- Windows Task Manager, page 3
- Windows perfmon Utility, page 5

RPL Analysis

In simulations that use RPL sets (e.g., RBS ruleset, optimization goal sets, expression slot set, initialization rules set, user defined accounting method set, iterative MRM ruleset or global functions set), the RPL Set Analysis tool can be used to identify functions that execute an inordinate number of times or spend a lot of time in their execution. See “RPL Analysis Tool” on page 101 for details.

RPL Timing and Count Predefined Functions

The following RPL predefined functions measure performance:

- RunTime() returns the number of seconds which have elapsed since the run began. See “RunTime” in *RiverWare Policy Language (RPL)* for details.
- DispatchTime() Returns the accumulated time spent executing object dispatch methods since the beginning of the current run. See “DispatchTime” in *RiverWare Policy Language (RPL)* for details.
- DispatchCount() returns the number of object dispatch method executions that have occurred since the beginning of the current run. See “DispatchCount” in *RiverWare Policy Language (RPL)* for details.

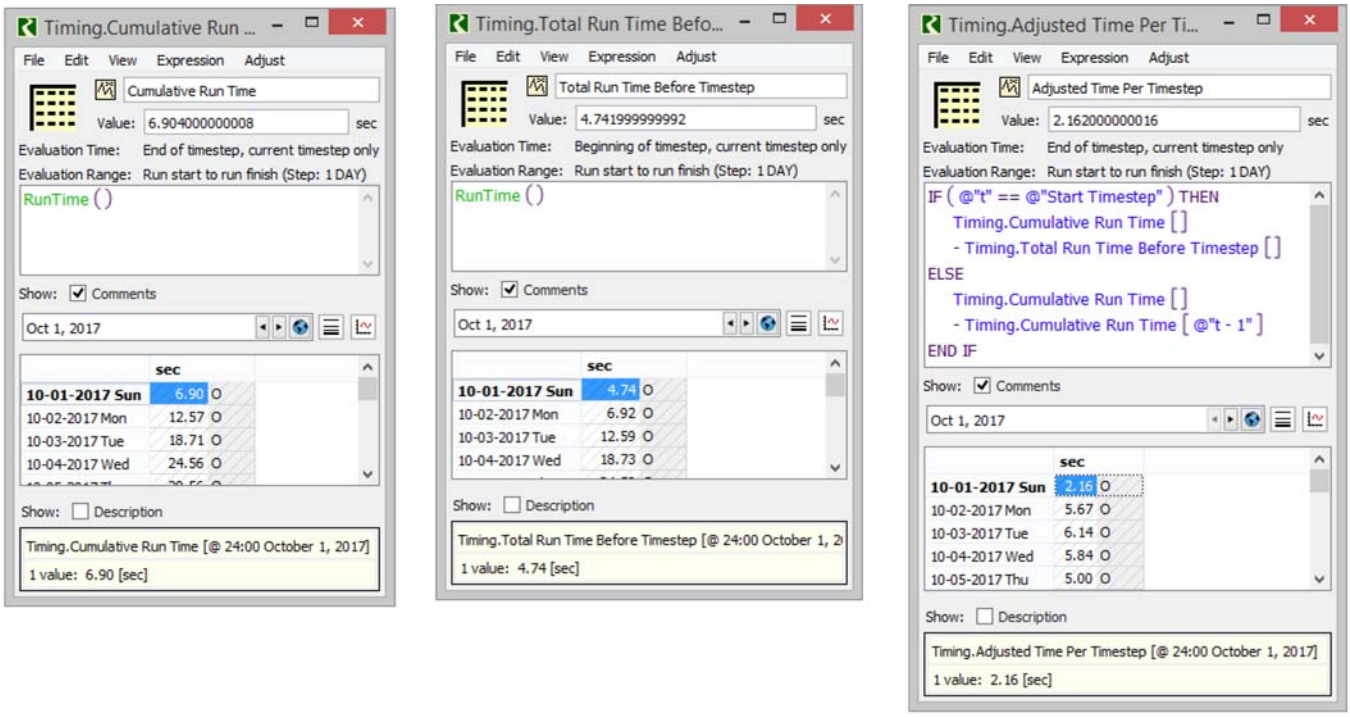
  Note: More specific dispatch times and counts are also available in the RPL Set Analysis tool; see “RPL Analysis Tool” on page 101.

Using expression slots or rules, these function can be used to record the performance. Following is a series of expression slots that record the cumulative number of seconds that have elapsed since the beginning of the run or determine the number of seconds each timestep takes. This can be used to determine if a particular operation is

taking more time than expected. For example, does a timestep that has a flood control operation take significantly longer than a non-flood control timestep?

Figure 1.1 presents an example of Expression Slots showing how the RunTime function can be used. The left and right functions are configured to evaluate at “end of Timestep, current timestep only”. The center function is configured to evaluate at “Beginning of Timestep, current timestep only”. Together, these give the “Time per timestep” shown in the expression on the right.

Figure 1.1



Windows Task Manager

The Task Manager is accessed by right-clicking the toolbar and selecting **Task Manager**. The Task Manager has two tabs that record performance information. The **Performance** tab displays a number of useful items as shown in Figure 1.2 and Figure 1.3.

- Total CPU usage
- Number of processors on this machine and a plot of their usages
- Total physical memory (RAM) on the machine. In Figure 1.3, it is approximately 32 GB of RAM.

The **Processes** tab displays the percentage of CPU and memory used for each process.

We will refer to this utility in future sections. When using this utility to assess the performance of a RiverWare model, make sure that other large processes are not running.

Figure 1.2 Task Manager CPU view

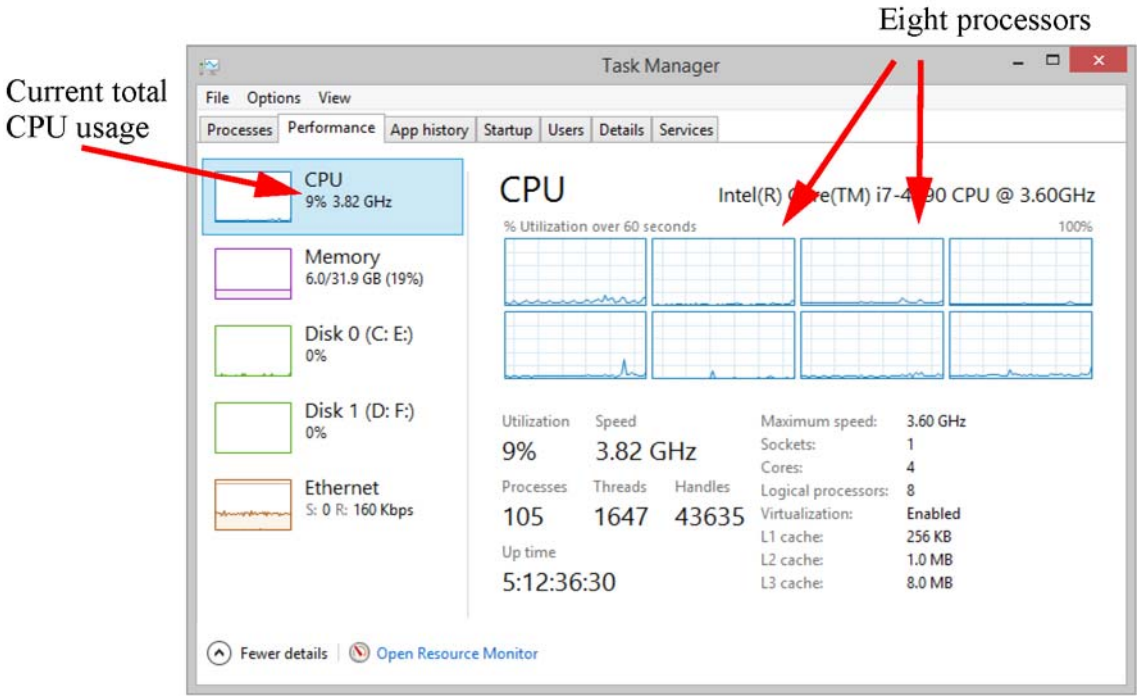
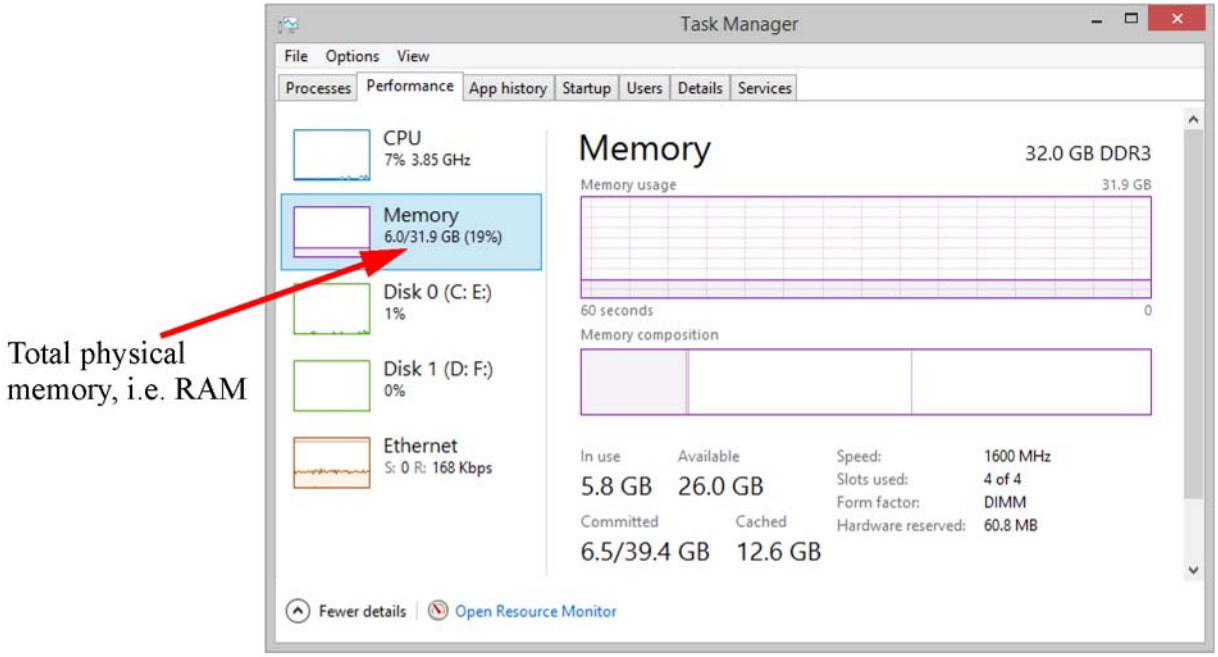


Figure 1.3 Task Manager Memory view



Windows perfmon Utility

In Windows, there is a performance monitoring utility called perfmon. This tool can be used to determine how processes including RiverWare are using memory. When using this utility to assess the performance of a RiverWare model, make sure that other processes are not running.

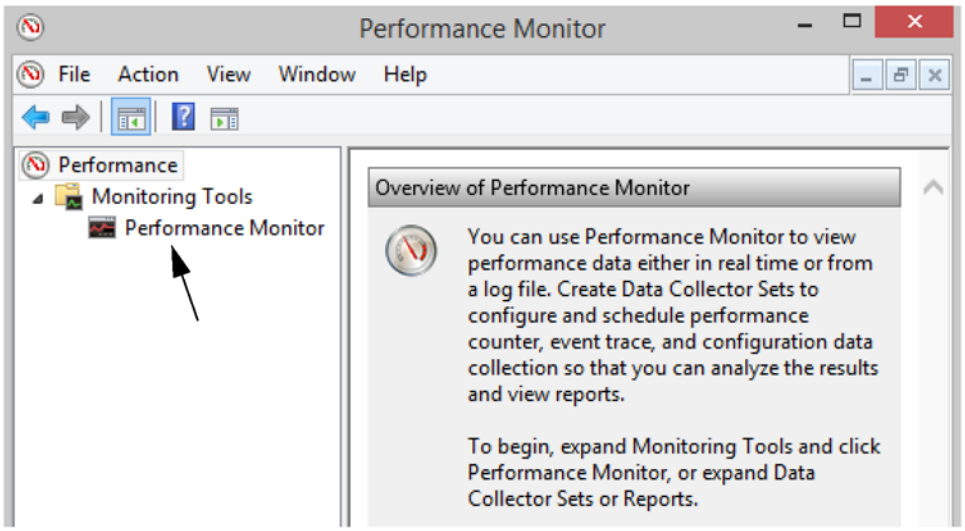
Use one of the following methods to start perfmon:

- Select **Start**, then **All Programs**, then **Command Prompt** and enter `perfmon`.
- Select **Start**, then **Run**, then enter `perfmon`.

The perfmon utility displays counters of variables. The following instructions are provided to show you how to use perfmon for simple analysis. This document is not comprehensive; use the perfmon Help for more information.

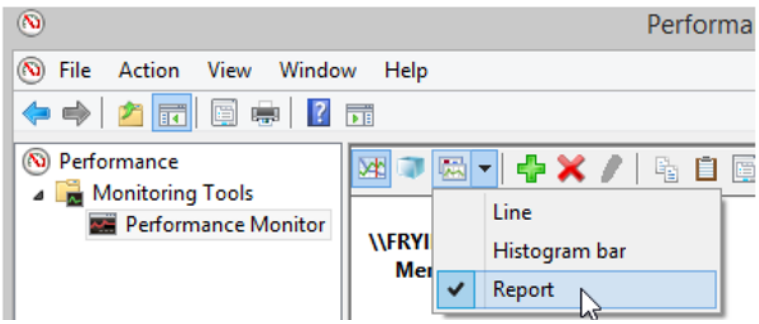
When you first start perfmon, a graph of the selected or default counters is provided, as shown in Figure 1.4. Select **Performance Monitoring**.

Figure 1.4 perfmon default view



The most useful view is the Reports view. Select the **Reports** icon to switch views. Figure 1.5 shows the Reports view.

Figure 1.5 perfmon Reports view



There may be counters defined. To add counters, select the **Plus** icon + to open the Add Counters dialog. Figure 1.6 illustrates.

Use the following steps in the Add Counters dialog:

1. Select a Performance Object and expand the list.
2. Highlight the counter from the list.
3. Select the instance, i.e. riverware.
4. Select **Add**.
5. Repeat for other counters.
6. Select **OK** when finished.

Figure 1.6 perfmon Add Counters dialog

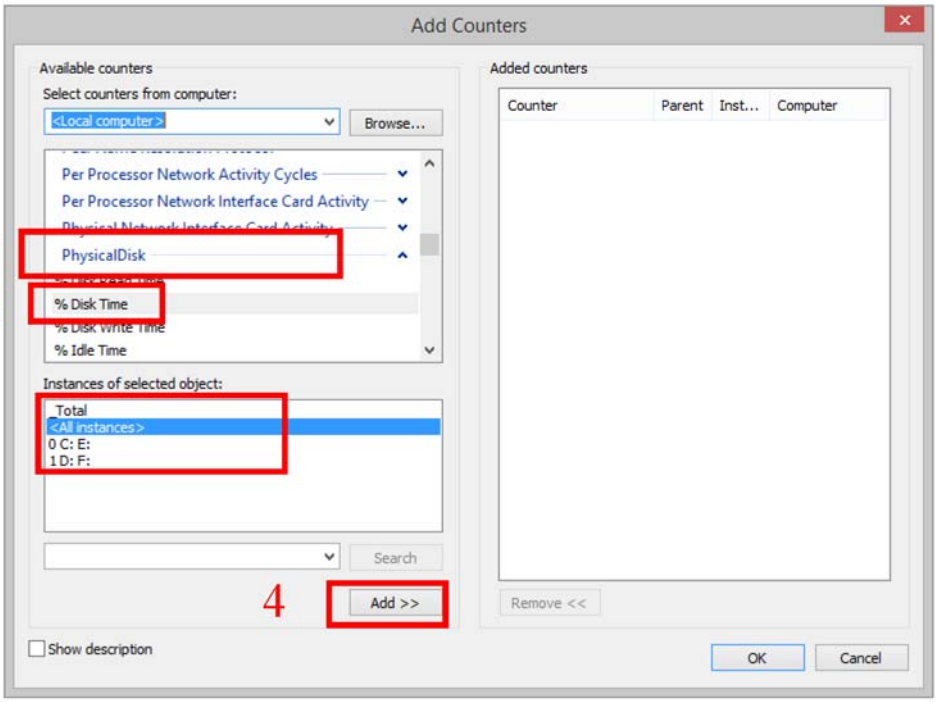


Table 1.1 describes the most useful counters.

Table 1.1

Performance Object	Counter	Instance	Description
Memory	Pages/Sec	NA	Hard Page Faults per second
Physical Disk	% Disk Time	Disk that contains the paging file	Percentage of time the selected disk is reading or writing.
Process	Virtual Bytes	_Total	Total size of all process
Process	Virtual Bytes	riverware	Total size of the RiverWare process

Table 1.1

Performance Object	Counter	Instance	Description
Process	Working Set	_Total	Size of memory that was recently touched by all processes
Process	Working Set	riverware	Size of memory that was recently touched by RiverWare

> **Note:** For more information or a description of any of these counters, highlight the counter in the Add Counter dialog and click **Show Description**. Once the perfmon tool is configured, it can be saved as a .msc file and reloaded as necessary.

Now that we have tools to collect performance information, we present recommendations for acting on this information to improve performance.

Run RiverWare More Efficiently

This section describes steps that can be taken to run RiverWare more efficiently and thus improve runtime performance.

Reduce Diagnostics Processing

When diagnostics are enabled, RiverWare must process diagnostics as well as perform the primary calculations of the simulation. The overhead associated with diagnostics consists of the following:

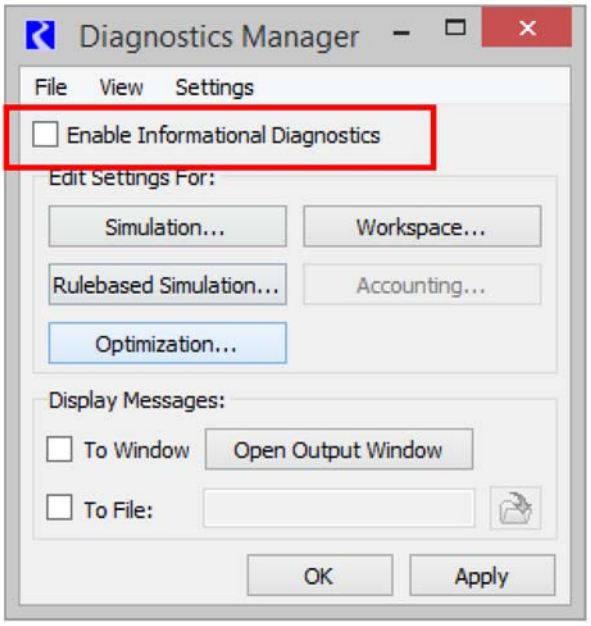
1. Decide whether individual diagnostics should be issued.
2. Collect the data to be posted.
3. Post the diagnostic.

In some contexts, Step 1. and Step 2. can require significant computation, but in most situations it is Step 3. that dominates diagnostic processing time. See “About RiverWare Diagnostics” on page 33 for a general discussion of diagnostics.

Disable Diagnostics

Since the overhead associated with processing diagnostics can represent a significant fraction of a run, if they are not necessary, and performance is an issue, then it is best to disable them. To disable, make sure the **Enable Informational Diagnostics** option is *not* selected in the Diagnostics Manager.

Figure 1.7 Diagnostics Manager with diagnostics disabled



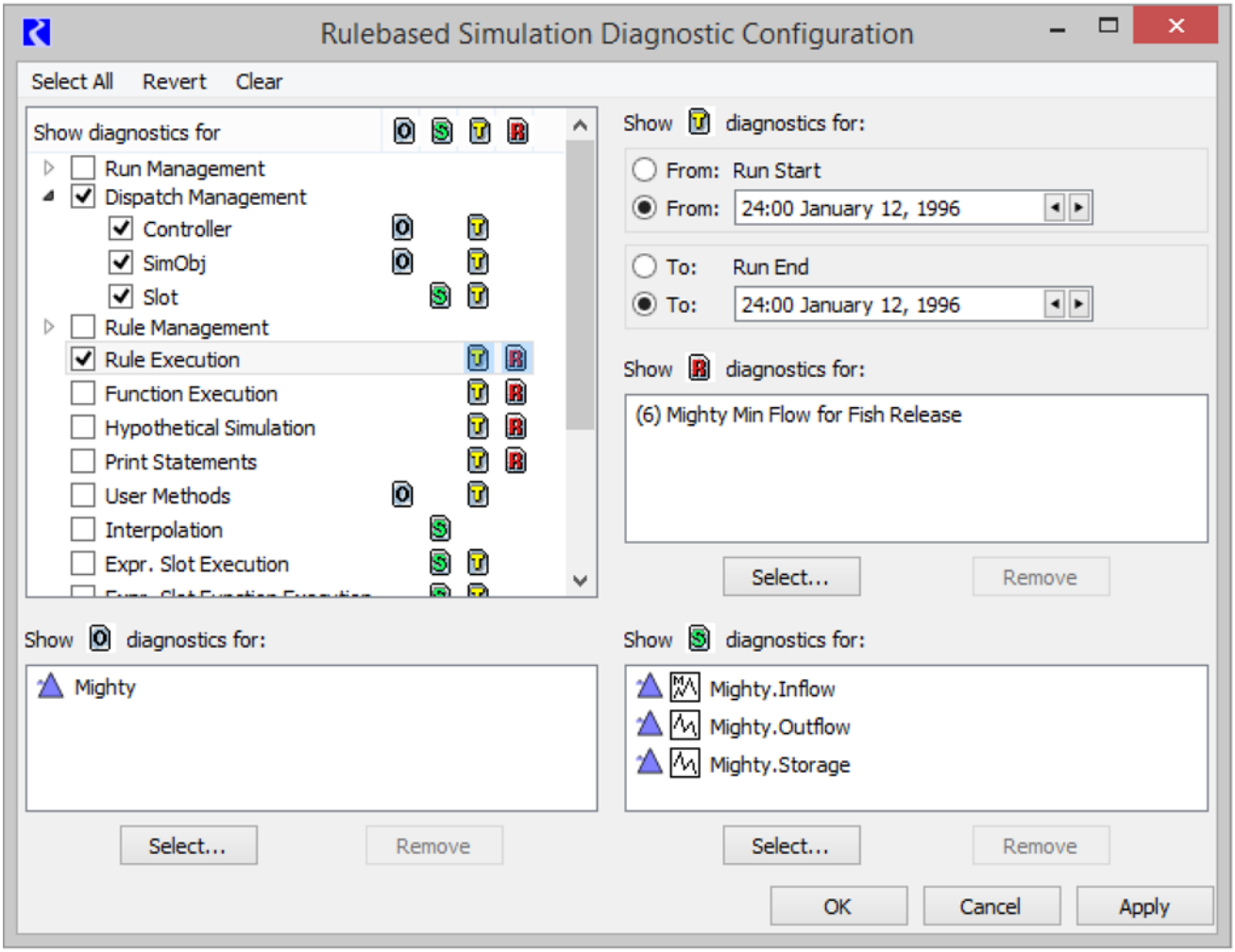
Filter Diagnostics

When diagnostics are being sent to the Diagnostic Output Window, the time to post a diagnostic is related to the number of diagnostics in the diagnostic output window. In other words, the more diagnostics that have already been posted, the more time it requires to post an additional diagnostic. This means that the time spent posting diagnostics within a total run is proportional to the square of the number of diagnostics posted. Thus even though posting an individual diagnostic is not expensive, it can take a lot of time to post a large number of them. By filtering diagnostics to post only those diagnostics currently of interest, run times are decreased and less effort is required to sift through diagnostics for the desired information.

Figure 1.8 shows the diagnostics filtering dialog with the following filters enabled:

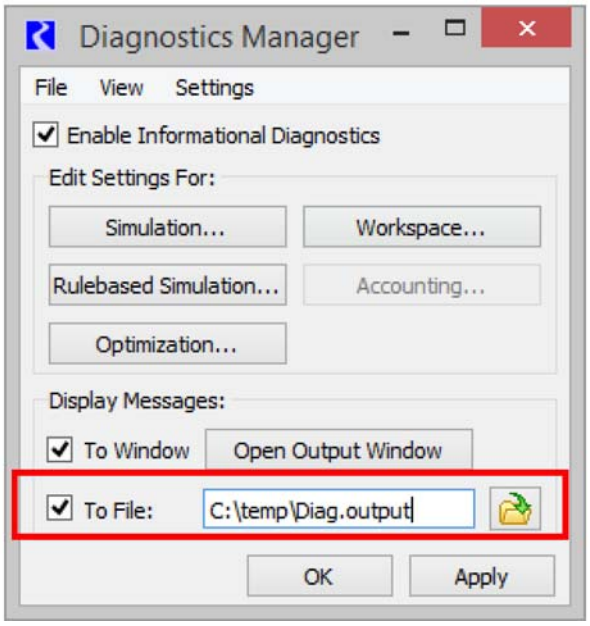
- Timestep = January 12, 1996
- Rules = Mighty Min Flow for Fish Release
- Object = Mighty
- Slots = Mighty.Inflow, Outflow and Storage

Figure 1.8 Diagnostics filtered



Send Diagnostics to a File

Figure 1.9 Diagnostics Manager indicating to send diagnostics to a file



If it is useful to see many diagnostics, consider sending them to a file instead of to the Diagnostic Output Window. File I/O is relatively fast, and one can still view the diagnostics in a text editor, perhaps one with more sophisticated searching and filtering capabilities than those provided by the Diagnostic Output Window.

> **Note:** Posting a diagnostic to the diagnostic output window requires additional memory usage, so sending diagnostics to a file instead of a window will reduce RiverWare memory use, which can have a positive impact on run times as well.

Break Long Runs Into Multiple Runs

For most models, the size of the RiverWare process is dominated by series data. If the model is too large to fit into virtual memory, there are two ways to break up the model, either in time or in space. To break the model up by time, divide the total run into multiple runs of shorter duration, configure DMIs to exchange data between runs, and write RCL scripts to conduct these runs in sequence. See "Batch Mode and RiverWare Command Language" in *Automation Tools* for details about RCL scripts.

Another possibility is to simulate upstream portions of the model for the full time range of interest and then import the results into a model that simulates the remainder of the basin. Of course, operational dependency of upstream objects on downstream objects may prevent this from being an option.

Close Unused Windows

Updating the graphical user interface during a run requires CPU time and increases the size of the RiverWare process. When it is desirable to run RiverWare interactively, you can reduce the amount of time updating dialogs

during the run by closing them. This is especially important for series data such as that displayed by an SCT, a SeriesSlot dialog or a model run analysis dialog. Each time a slot value changes internally, the corresponding slot's dialog might need to be updated if it is open. In general, if performance is critical, we recommend that all unused dialogs be closed.

Run in Batch Mode

Consider completely avoiding updating the user interface during a run by writing a RCL script to conduct a batch mode run. After a run, you can then bring up RiverWare interactively to analyze the results.

Improve Your RPL Set Logic

Although RiverWare often spends much of its time dispatching, RPL statement execution can represents a significant factor of the time spent during a rulebased simulation. Some RPL set logic is more computationally intensive than others, and changing RPL logic with this in mind can often improve model run times. Use the RPL Analysis tool to identify functions that are taking a long time or execute too many times (see "RPL Analysis Tool" on page 101). Then, determine if there is a more efficient way to write these functions. See "Developing Efficient RPL Expressions" in *RiverWare Policy Language (RPL)* for a list of suggestions to writing efficient RPL expressions.

Invest in Hardware

Often, the most significant improvements in RiverWare model performances can be obtained by investing in hardware. When investing in hardware, the following should be considered (in order of most significance):

- Physical Memory (RAM) - purchase enough to eliminate paging
- If completely eliminating paging is impossible, purchase multiple faster disks.
- If making multiple runs, purchase additional machines or use multiple CPUs.
- Purchase a machine looking at the following attributes:
  - Memory size
  - Cache size
  - Bus speed
  - Disk speed
  - CPU speed

Install More Physical Memory (RAM)

If your model is causing paging, then installing more RAM will lead to less paging and faster run times. Most modern operating systems allow programs to access more memory locations than are available to that program in physical memory. When RiverWare attempts to access a location that is not in physical memory, a page fault occurs and the operating system brings the page containing that location into physical memory. Since the time to access a

memory location is likely to be orders of magnitude quicker if that memory location is currently in physical memory than when it is located in disk storage, reducing the number of page faults during a RiverWare run will decrease the run time.

To determine if your model leads to page faults and the associated disk I/O delays, use the Windows perfmon tool to display the Memory: Pages/Sec counter and the PhysicalDisk: % Disk Time for the disk which contains your system page file. See "Windows perfmon Utility" on page 5 for details.

These counters will be non-zero when the system is processing (hard) page faults. You will also notice that when RiverWare is doing significant I/O, the CPU Usage for the RiverWare process as displayed in the Windows Task Manager will be less than full (for a single CPU, less than 100%). Figure 1.10 shows a sample model run.

Note: RiverWare Virtual Bytes is 11.1 GB.

Figure 1.10 Sample model run

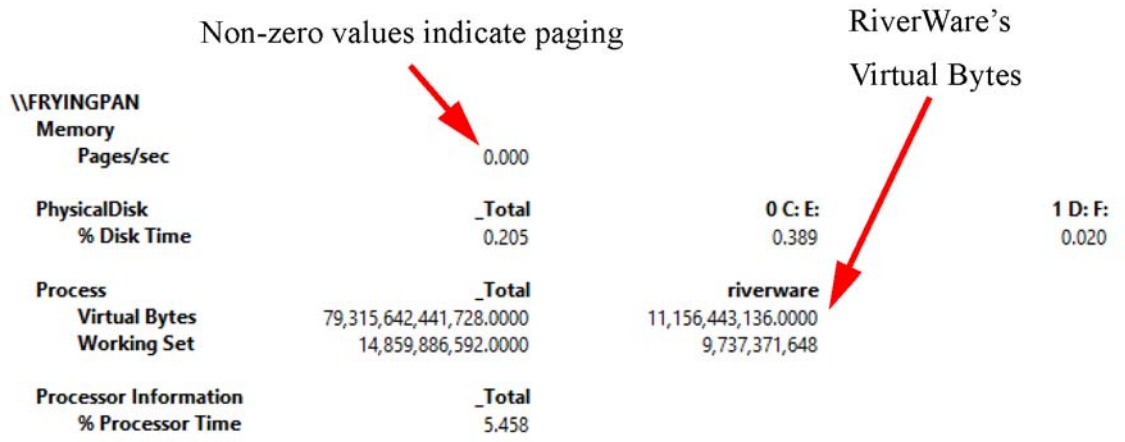


Figure 1: Using the perfmon to identify if paging is occurring

How much RAM do you need? If your model/computer consistently utilizes 11 GB of memory, then make sure you have at least 12 GB of RAM available for it, and preferably more. Adding enough RAM will ensure that all the required memory used will indeed be used from the RAM, and not supplemented by paging. To determine how much memory is used during a model run, look at the perfmon utility. Under the Process, Virtual Bytes entry, it shows the number of bytes used by the riverware process. This is the total space that RiverWare is using. To fit RiverWare completely in memory, the amount of RAM should be larger than this plus enough for the operating system.

Purchase Additional, Faster Disks

If is impossible for you to eliminate paging during a model run, one strategy is to make the paging faster. To do this, purchase faster disks so the paging operation is faster.

Note: This refers to physical disks, not Windows *drive letters*, which are really partitions on physical disks. Partitioning a single drive does not improve performance.

It also helps if the different operations running on the machine do not compete for use of the same disk. For this reason it helps to have your paging files on different physical disks from your normal data and programs. See "Adjust the Size or Location of Your Paging File" on page 15 for details.

Use Additional CPUs for Multiple Runs

If you are making multiple runs, i.e. alternative analysis, and you have access to multiple machines (or a single machine with multiple CPUs), then you can run multiple RiverWare processes simultaneously. For example, if you are completing an alternatives analysis and are ready for three production runs, use three machines to run alternatives 1, 2 and 3 in parallel. The entire analysis will take the same time as running one run and should be three times faster than running them in series. Batch processing can help to manage many parallel different runs; see "Batch Mode and RiverWare Command Language" in *Automation Tools*. Also, see "Distributed Concurrent Runs" in *Solution Approaches* for details about running traces of a Concurrent Multiple Run in parallel on different processors on the same computer.

Also remember that a machine with multiple CPUs will speed execution of parallel runs but the total physical memory is shared among all CPUs and can limit run speed. In that regard, it may still be better to run parallel runs on completely separate machines.

Purchase a Bigger, Faster Machine

Above we have presented some approaches to upgrade existing hardware so that RiverWare performs better. Often it may be more cost effective to replace hardware with new machines. When selecting a new machine look at the following attributes, in order: physical memory size, cache size, bus speed, disk speed, and CPU speed.

Physical Memory

Purchase as much physical memory (aka RAM) as you can. Having plenty of physical memory allows you to "fit" all computations and data allocations into memory and prevent disk paging. Typical RAM configurations are 8GB all the way up to 64GB or more. For example, training machines at CADSWES have 8GB of RAM while development machines utilize 64GB of RAM.

Most operating systems configured for "servers" do allow large banks of main memory to be used. At CADSWES we have not yet experimented with Windows server configurations, so we cannot attest to the potential speed benefits of purchasing such a machine, nor can we guarantee that there will not be attendant problems. This is an option worthy of consideration.

Cache Size

Data accesses are faster if the data reside in memory than if they reside on disk. Modern computers further speed up memory accesses by employing caches of faster memory. Cache memory is physically different from main memory, making it more expensive, and cache memory physically resides closer to the CPU (the heart of the computer) than does main memory.

Thus, if the data needed resides in a memory cache, the access will be faster than if it resides in main memory. Consequently, you will benefit from having the largest caches available on the machines you purchase. Some

architectures have Level 1 (L1), Level 2 and Level 3 caches. Some have only L1 and L2 and often only the lower-level (L2, L3) cache sizes are configureable when you purchase a machine.

Front-side Bus Disk Speed

Accesses to main memory are slower than accesses to cache memory. This is because the cycle time of the cache memory is lower than that of main memory, but also because the electrons have to travel farther from main memory than from cache memory. On many architectures, data (and instructions) flowing from main memory to the CPU have to travel across a central system bus. Consequently, the speed of the bus determines the speed of main-memory accesses, and a good rule of thumb is to purchase the fastest bus you can get. Most Intel-based computers call this bus the “Front-Side Bus”.

CPU

Most object-oriented applications, RiverWare included, are not limited by CPU speed, due to the nature of their memory access patterns. Consequently, CPU speed is not the most important factor in purchasing a new machine as most new machines have adequate CPU speeds.

One of the ways modern computers are made more powerful is by giving them more than one CPU. Some applications are *multi-threaded*, which allows them to run somewhat faster by using more than one CPU in parallel. RiverWare is not multi-threaded, however, you might still benefit from having multiple CPUs because the operating systems typically run more than one program, even if you have nothing but RiverWare running. Having another CPU can reduce the amount of time that RiverWare has to give up the CPU and its cached data for use by another process. If you tend to use your machine for things other than running RiverWare (that is to say, you do not have a dedicated machine for RiverWare), then you most certainly can benefit from additional CPUs.

> **Note:** In so-called “multi-core” machines, two or more CPUs might share some cache, but be unable to share all of the cache. For example, a four-core machine might in fact be two two-core machines, where two CPUs share half the L2 cache and the other two CPUs share the other half of the L2 cache. This kind of architecture might be labeled as having, say 8 MB of L2 cache, when in fact it has two 4 MB L2 caches (2 x 4 MB). In this example, the most L2 cache that RiverWare could use is 4 MB. This also means that a process scheduled on the CPU that shares cache with the RiverWare CPU can have a negative impact on RiverWare performance, whereas the processes scheduled on the other CPUs have no effect on RiverWare available cache. The upshot: a two CPU machine with caches dedicated to each CPU might perform better than a two CPU machine with shared cache, all else being equal.

Manage Memory

When you run a RiverWare model on a computer, there are other applications that are competing for the computer’s memory resources (such as the CPU or a disk). If your model is small and your machine is “big/fast”, there is no conflict. If the opposite is true, your model is big and your computer is “small/slow”, (this is more likely if you are reading this), then there may be competition between applications for the limited system resources. This section describes two approaches to make the RiverWare model run faster by adjusting other applications or changing the configuration of the operating system.

Close Other Applications

When other applications are running at the same time as RiverWare, it is possible that competition for system resources will degrade RiverWare performance. The obvious solution is to close the other applications. But, if you do not want to close out of other applications, it is possible to tell the operating system to give preference to RiverWare by increasing its priority. To change the priority:

On the Task Manager, select the **Processes** tab. Locate the riverware.exe process, (note, there may be more than one if you are running multiple models). Right-click riverware.exe and select **Set Priority**, then **High**. It will warn that system instability may occur. Select **OK**.

Adjust the Size or Location of Your Paging File

If RiverWare is paging and you cannot reduce the amount of paging, then you can sometimes configure your system to reduce the overhead associated with paging. First, move the page file off the disk that holds your system partition to another fast, dedicated hard disk. By moving your page file to a dedicated disk, you can reduce the time to write and read pages. In addition, if you can spread the page file across several non-system disks, you can decrease page file competition.

It is beyond the scope of this discussion to provide all the potentially relevant details for various systems configurations, however your system administrator or IT department should be able to assist if necessary. For other resources, search the internet for “windows page file”.

Chapter 1
Model Run Performance

Chapter 2
Model Run Analysis

Topics

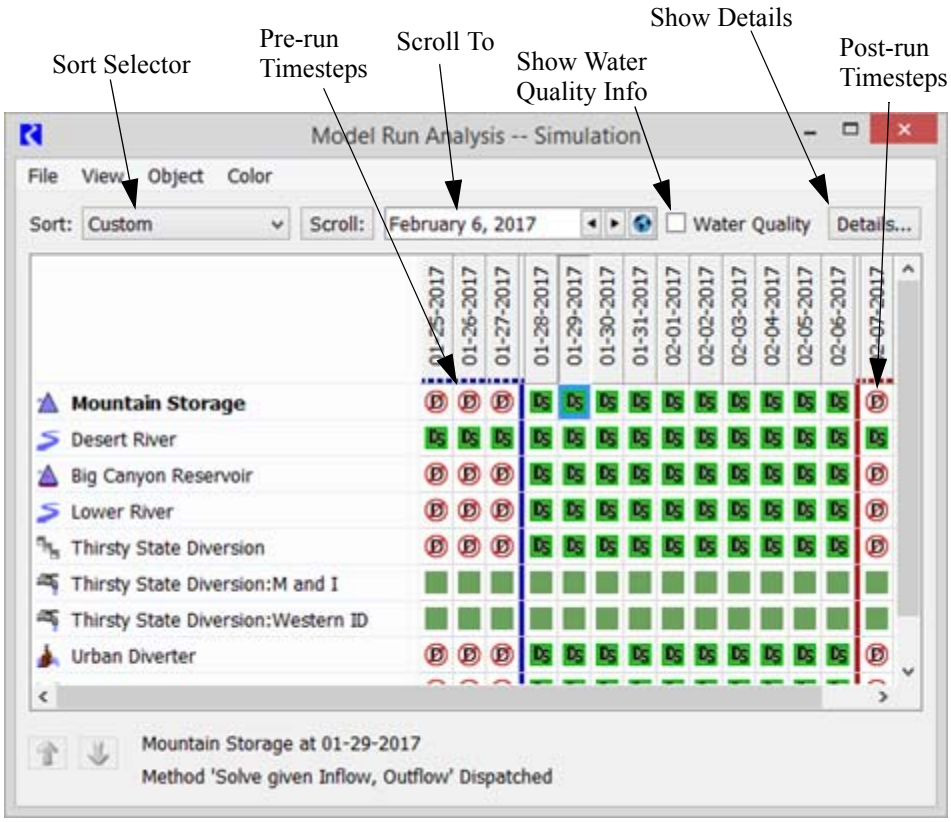
- Simulation, page 17
- Rulebased Simulation, page 25

Simulation

The Model Run Analysis utility provides post-run diagnostics regarding object dispatching and known values of dispatching slots at every timestep of the simulation. The Model Run Analysis dialog is invoked by selecting **Model Run Analysis** from the **Utilities** menu or by selecting the **Run Analysis** shortcut button on the toolbar. 

The Model Run Analysis tool provides dispatch information at two levels of detail. At the workspace level, the dispatch state of each object for each timestep of the model run is visible. At a more detailed level, the tool allows an object's dispatch methods and known slot values at the end of each timestep to be viewed. Figure 2.1 shows a sample Model Run Analysis dialog.

Figure 2.1



The Model Run Analysis—Simulation dialog displays the workspace-level view of dispatch conditions. This dialog consists of a two-dimensional matrix with workspace objects on one axis and model run timesteps on the other axis. The left-most part of the matrix, separated by a blue vertical line, indicates prerun timesteps. Post-run timesteps (when configured) are shown separated by a red line. Each cell in the matrix shows the dispatch state of an object at the indicated timestep. Dispatches are registered automatically during model runs; dispatch information is displayed when available. Because they never dispatch, Data Objects and Snapshot Objects are not included in the list of Objects.

An Object has one of four possible dispatch states:

- **Object has not dispatched**. The known and unknown slots for the timestep did not meet the requirements of any method on the dispatch table for the Object.
- **Object has dispatched but not solved**. Conditions of a single method on the dispatch table were met; the method dispatched, but the Object did not solve completely. (in yellow)
- **Object has fully dispatched**. Conditions of a single method on the dispatch table were met; the method dispatched, and the Object solved.

- **Object has no dispatch conditions**. The Object has no entries on its dispatch table, the Object has no dispatch conditions (e.g. Gage Object or Thermal Object), or the model has not been run.

Model Run Analysis Dialog

Following is a tour of the model run analysis dialog.

Sorting

The object list can be sorted using a variety of methods, as follows.

Sort Menu

The order in which Objects appear in the dialog may be rearranged from its original alphabetical order. The Sort pull down menu allows the user to sort the objects by:

- **Custom**. The user may define a custom order using the **Up** and **Down** arrows.



  Initially, the objects are arranged by the internal ordering of the objects in the model file, as defined by the order in which objects were created.
- **Name**. Alphabetical
- **Type**. Arrange by the object's type. The types are then arranged alphabetically.
- **Position**. Arrange using the workspace layout to arrange the objects top to bottom based on y-coordinate (This assumes objects in the basin go top to bottom and the Run Analysis tool is displayed "Time on Top")
- **Custom from Workspace**. Use the custom order created on the Workspace Object List. See "Simulation Object List" in *User Interface* for details.



Move Object Up and Down buttons

The **Up** and **Down** arrows are used to move one or more selected objects up or down in the list. These buttons are only available for the Custom sort order. When using any of the other sort orders (Name, Type, Position) the **Set Order** button is displayed and active. This button sets the currently displayed order as the new Custom order, switches the sort menu to Custom, and enables the **Up** and **Down** arrows. A warning message dialog is presented to alert the user that the custom order will be overwritten. This functionality allows the user to first sort by one of the common approaches, save the order as custom, then make detailed changes using the **Up** and **Down** arrows. Any further changes to the Custom order are saved automatically.

Scroll To

The **Scroll To** button allows the user to enter a date or use the datetime spinner to select a date. Selecting **Scroll** moves the dialog to that date.

Water Quality Dispatch Information

When water quality is enabled, the **Water Quality** checkbox is displayed on the main grid dialog and the **WQ** checkbox is shown on Dispatch Details dialogs. When enabled, the Water Quality dispatch information is shown instead of the water quantity dispatch information. See "Viewing Water Quality Dispatch Information" in *Water Quality* for details.

Time on Top and Time on Side

The Model Run Analysis tool can be shown with either time on the top or time down the side. In the **View** menu, the chosen option will have a check mark next to it.

Rules Grid

Even for a simulation model, the user can select to show the Rules Grid from the **View** menu. This can be useful when switching between the simulation and rulebased simulation controllers. Showing the Rules Grid provides additional information on the priority of the rules that set slots. See "Main Analysis Dialog" on page 25 for details.

Show Object Icons

In the **View** menu, the user can select **Show Object Icons**. When checked, the icon for each simulation object is shown in on the tool. This option is only available when shown in the Time on Top configuration. For the Time on Side configuration, object icons are never shown.

Update After Every Timestep

In the **View** menu, the user can select the **Update After Every Timestep** option. If this option is selected, the utility, when opened, will be updated after every timestep. For models with many objects or timesteps, this processing can be very time consuming and lead to adverse performance. For large models, it is advisable to turn this option off or keep the Model Run Analysis utility closed until the model has stopped running.

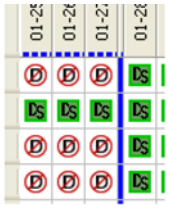
> **Note:** The utility is always updated when the run stops. For small models, this option can remain checked.

Fonts

The fonts used in the Model Run Analysis dialog can be changed from the **View**, then **Select Fonts** menu. Once a non-default font is configured, it is preserved as the *selected font*. By selecting **View**, then **Use Default Font** and **View**, then **Use Selected Font**, the user can switch between the default and specified font.

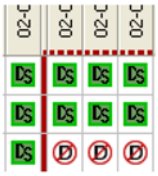
Pre-run Timesteps

Timesteps before the start timestep are displayed with a blue dotted line below and to the right of the timestep label. Also, there is a blue separator line between the initial and start timestep. Only certain objects (reaches, confluences, control points, and stream gages) dispatch during prerun timesteps,. Thus, most objects indicate they have not dispatched during the prerun period.



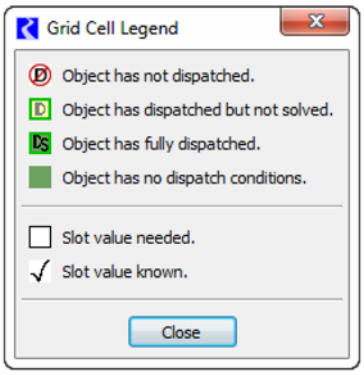
Post-run Timesteps

Timesteps after the finish timestep are displayed with a red dotted line below and to the right of the timestep label. Also, there is a red separator line after the finish timestep. Post-run dispatching is enabled on the run control parameters. See "Simulation Run Parameters" in *User Interface* for details.



Legend

A legend used in the Model Run Analysis can be displayed by selecting **View**, then **Grid Cell Legend**.



Dispatch State

Selecting an Object or timestep cell displays its dispatch state in text form at the bottom of the Dispatch Information dialog. If the Object dispatched at that timestep, then the name of the method under which it dispatched is also displayed.

Context Menus

On the Model Run Analysis, right-click context menus are available to **Show Dispatch Behavior Details**, **Show Special Results Details, Open Object**, **Disable Dispatching**, **Copy Time**, and **Global Scroll**.

Disabling Dispatching

The Model Run Analysis dialog may be used to disable or enable dispatching for individual Objects. Disabling dispatching prevents the object from dispatching at any timestep, regardless of what slots are known. The Object's Beginning of Run behavior, however, may never be disabled. This functionality is useful to debug models. To disable dispatching for an Object, select its name and select **Object**, then **Disable Dispatching** from the Model Run Analysis dialog menu bar. The Object name is shaded with red diagonal lines to indicate that it will no longer dispatch. Also, the object icon on the workspace and on other dialogs is dimmed. See "Enabling and Disabling Dispatching" in *User Interface* for details. To reenable dispatching, select the name of the Object for which dispatching has been disabled and select the **Object**, then **Enable Dispatching**.

Details Dialogs

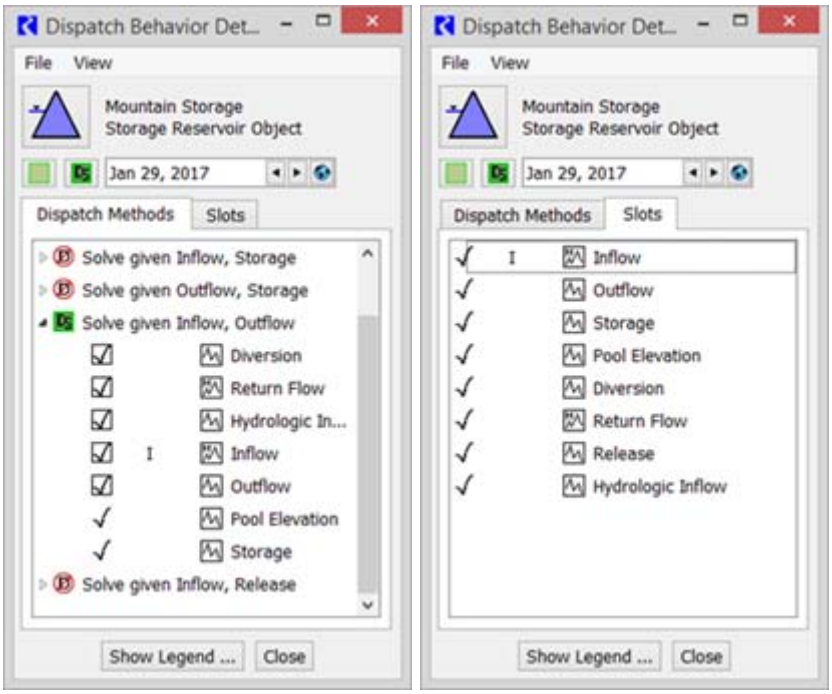
From the Model Run Analysis tool, you can see additional information for any object at any timestep. The information available depends on the model's controller and other selections.

Dispatch Behavior Details Dialogs

The Dispatch Behavior Details dialogs allow the entries in the dispatch table and the known slots at the end of each timestep to be viewed for each object. To open a Dispatch Behavior Details dialog, select and open a cell in the Model Run Analysis dialog that represents the desired object and timestep.

The Dispatch Behavior Details dialog has two views. The default view depends on the dispatch state of the selected object/time. Figure 2.2 shows a sample of each view.

Figure 2.2



If the Object has *no dispatch* conditions, or the model has not yet been run, a blank dialog is shown.

If the Object *did not dispatch* during the selected timestep, the Dispatch Behavior Details dialog appears showing the **Slots** tab. Each of the displayed slots is a required known and/or a required unknown for one of the Object's dispatch methods. The checkmarks indicate which slots were known at the end of the selected timestep.

To view the entries in the dispatch table (possible dispatch methods) for this object, select the **Dispatch Methods** tab. The Dispatch Methods view shows all of the methods in the Object's dispatch table, each marked with a symbol indicating whether the method dispatched. Selecting the tree view to the left of a dispatch method name opens a list of its dispatch conditions. Slots in the list with a checkbox to their left are required knowns for the method to dispatch. Slots with no checkbox are required unknowns. Checkmarks indicate which slots were known at the end of the timestep.

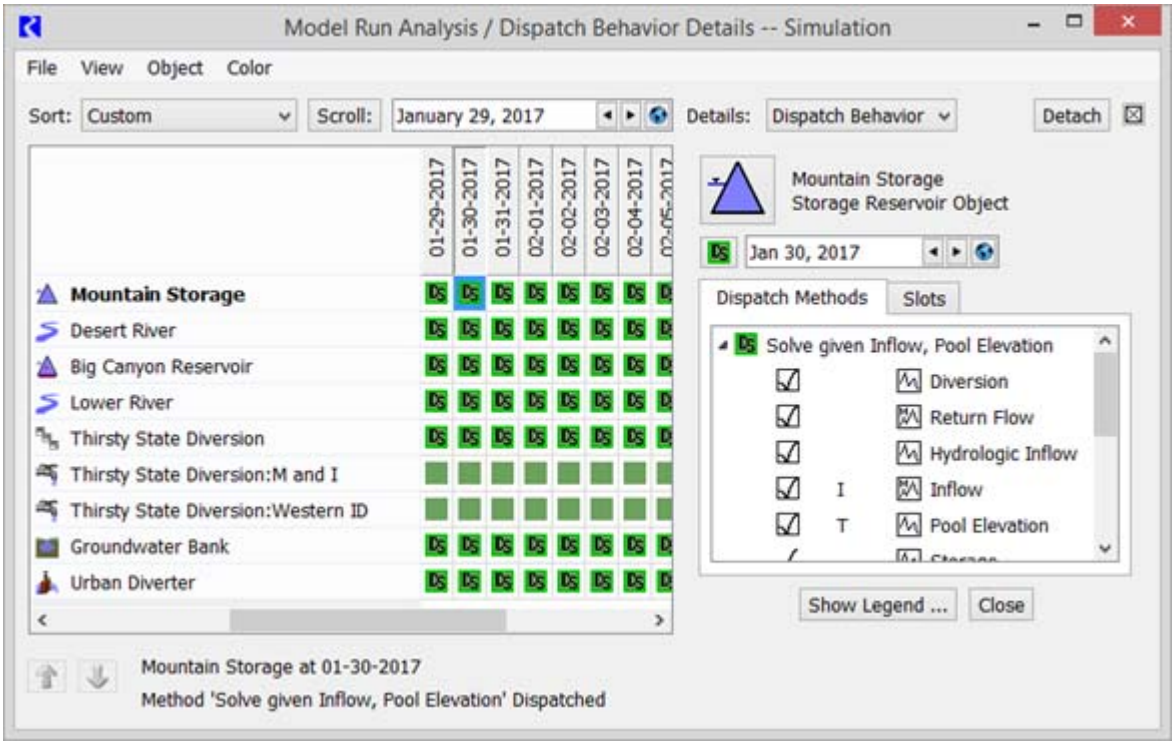
If the object *did dispatch* during the selected timestep, the Dispatch Behavior Details dialog opens in the Dispatch Methods view. The method which dispatched at the selected timestep is marked with the dispatched symbol and is opened to show its dispatch conditions.

> **Note:** All the slots are checked as known, even those required to be unknown. This occurs because the object solved correctly for all slots, making them known by the end of the timestep. We know that the dispatch conditions requiring unknowns were met at the time of dispatching because the method could not have dispatched otherwise. This may seem slightly confusing, but remember that the known/unknown status of particular slots is truly of interest only if the object failed to dispatch at the selected timestep. If the object did indeed dispatch, all of the conditions must have been met.

The Dispatch Behavior Details dialog contains a date selector which moves the detail view forward and backward in time. This is useful to determine how the set of known slots changes from timestep to timestep.

The Dispatch Behavior Details dialog can also be docked in the Model Run Analysis dialog. From the Model Run Analysis dialog, select **Details**. This adds the Dispatch Behavior Details dialog to the same window as the Model Run Analysis dialog. Figure 2.3 illustrates.

Figure 2.3



When docked, the dialogs can be separated using the **Detach** button. When in standalone mode, the Dispatch Behavior Details dialog can be docked from its **File**, then **Dock in Model Run Analysis** menu.

In the Dispatch Behavior Details dialog, selecting the **Object** icon brings up the Open Object dialog. Similarly, Selecting the icon  brings up the Model Run Analysis and docks the Dispatch Behavior Details dialog.

Rule Effects Details Dialog

The Rule Effects Details dialog displays the names of all rules and whether each rule succeeded at the selected timestep. See “Rule Effects Details Dialog” on page 29 for details.

Special Results Details Dialogs

The Special Results Details dialogs provide access to results for certain algorithms and computations. Currently the implemented special results are for USACE Methods. See “Model Run Analysis—Special Results Details Dialog” in *USACE-SWD Modeling Techniques* for details.

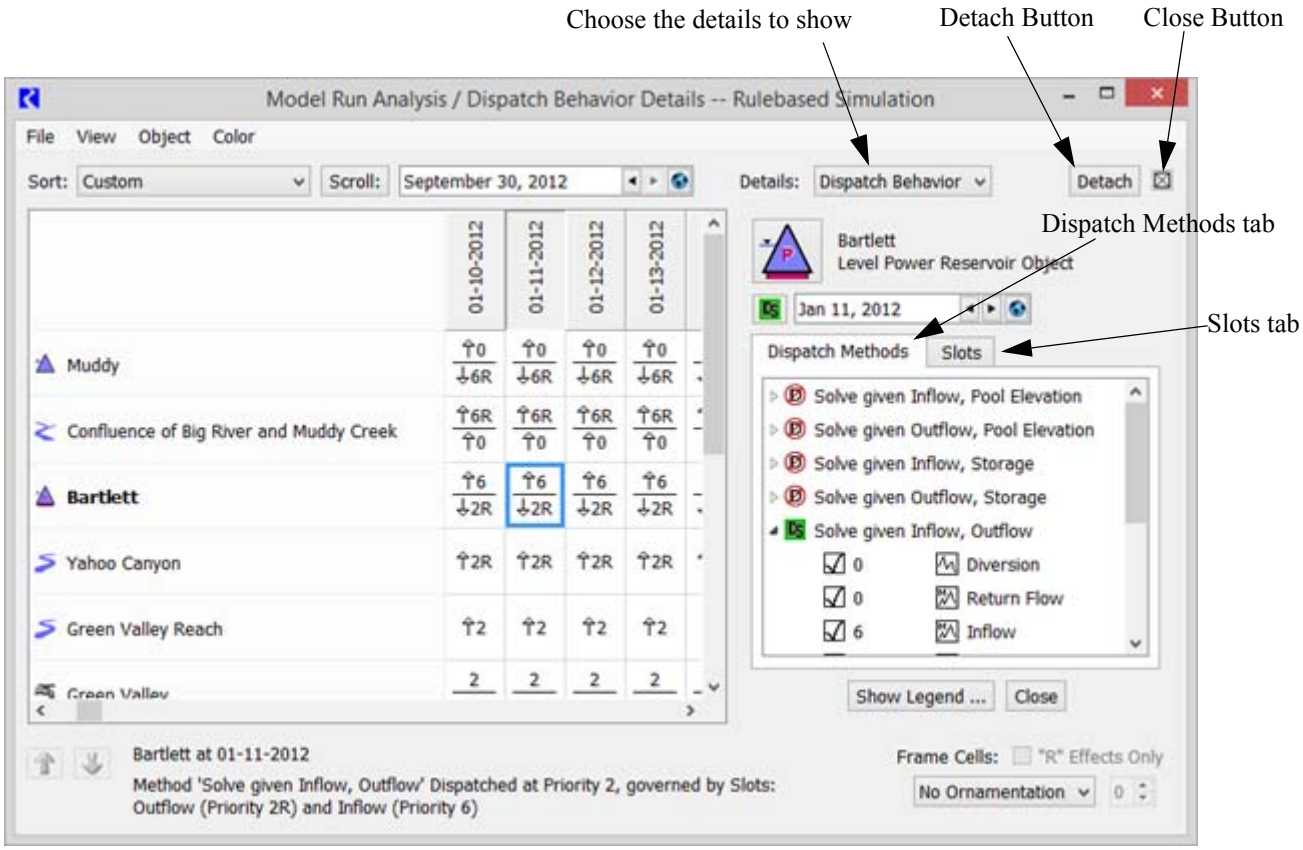
Rulebased Simulation

A modified version of the Model Run Analysis—Simulation dialog, which is available when the Simulation controller is active, is available when the Rulebased Simulation controller is active. The Rulebased Simulation version of the Model Run Analysis—Simulation dialog is called the Model Run Analysis—Rulebased Simulation dialog. It provides important information regarding slot priorities, determination of dispatch methods, and propagation of the effects of rules across a model. It is a diagnostic tool to be used in conjunction with the regular diagnostics to debug rules and verify that a run has solved as expected.

Main Analysis Dialog

The Model Run Analysis—Rulebased Simulation dialog is opened by selecting **Model Run Analysis** from the **Utilities** menu or selecting the **Model Run Analysis** toolbar button. Figure 2.4 shows a sample dialog.

Figure 2.4



For each object at each timestep, the cells provide information about the dispatch state of the object. The five possible dispatch states and their icons are as follows:

-  **Not Dispatched.** The known and unknown slots for the timestep did not meet the conditions of any method on the dispatch table for the object.

-  **Dispatched But Did Not Solve.** Conditions of a single dispatch method were met, the method dispatched, but the object did not solve completely (background is yellow).

-  **Dispatched and Solved Based Entirely on User Input Values.** Conditions of a single method were met from user input; the method dispatched, and the object solved completely.

-  **No Dispatch conditions.** The object has no entries on its dispatch table; the object has no dispatch conditions (e.g., Thermal Object), or the model has not been run.

-  **Dispatched and Solved Based on Highest Priority Known Values -** Conditions of a single method were met by prioritizing the slots, the method dispatched, and the object solved completely.

The first four dispatch states are identical to those in the Model Run Analysis—Simulation dialog. The last dispatch state is specific to the Model Run Analysis—Rulebased Simulation dialog. It provides information about the priorities of the governing slots for a dispatched object. The priorities and possible R flags of the governing slots are a good way to verify the solution of a Rulebased Simulation run. See "Simulation" on page 17 for details.

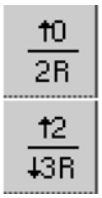
Recall that Reservoir objects have two governing slots, selected from the list of potential governing slots. Reservoirs may solve downstream due to Inflow and Storage or Inflow and Elevation, upstream due to Outflow (or Energy or Release) and Storage or Outflow (or Energy or Release) and Elevation, or it may solve for its ending Storage and Pool Elevation due to Inflow and Outflow (or Energy or Release). The intersection of the required knowns of the dispatch method and the potential governing slots of the object determine the governing slots. The governing slots shown in the Model Run Analysis—Rulebased Simulation dialog show us the direction of the solution.

When an object or timestep cell is selected, a box at the lower-left corner of the Model Run Analysis dialog displays its dispatch state in text form. If the selected object dispatched at the selected timestep, the box shows the name of the method with which it dispatched and the names and priorities of the governing slots.

The arrows next to the priorities in the cells represent the direction of water flow through the slot. Outflow is a downstream link to other objects in the model; it is marked by a down arrow. Inflow is an upstream link to other objects in the model; it is marked by an up arrow. The absence of an arrow next to some of the governing slots indicates that they are not topologically relevant. These are slots like Storage and Pool Elevation which are neither linked upstream nor downstream.

Example 2.1 Example:

Consider the following two cells. They describe two reservoir objects which are linked. The upstream reservoir is shown above the downstream reservoir. Can you reconstruct how this model solved?



Answer:

The upper reservoir dispatched given the following slots:

- Inflow (there is an **Upstream Linkable** arrow next to the slot priority), which was set as a user input (the priority is 0).
- Storage or Pool Elevation (there is no arrow next to the slot priority), which was set by Rule #2 (the priority is 2R).

Therefore, the upper reservoir must have solved for its Outflow at priority 2; Rule #2 must have been the last successful rule at the time the upper reservoir got enough information to solve.

The lower reservoir dispatched given the following slots:

- Inflow (there is an **Upstream Linkable** arrow next to the slot priority), which was set as a result of the upper reservoir's dispatch at priority 2 (the priority is 2).
- Outflow (there is a **Downstream Linkable** arrow next to the slot priority), which was set by Rule #3 (the priority is 3R).

So, the lower reservoir must have solved for its Storage and Pool Elevation at priority 2 or 3. We cannot be sure of the priority of Storage and Pool Elevation from the information given. If Rule #2 made its assignment first, the last successful rule to fire before the lower reservoir's dispatch must have been Rule #3; Storage and Pool Elevation would have a priority of 3. If, however, Rule #3 made its assignment first, the last successful rule to fire before the lower reservoir's dispatch must have been Rule #2 (which would cause the lower reservoir's Inflow to be known); Storage and Pool Elevation would have a priority of 2.

The directional arrows may be turned on or off by selecting or clearing the **Show Slot Link Directions** checkbox under the dialog **View** menu. In the same menu, you can enable or disable the automatic updating that occurs every timestep by selecting or clearing the **Update After Every Timestep** checkbox. Turning off the updating may save some time if you are running a long model with the Model Run Analysis dialog open.

Dispatch Behavior Details Dialog

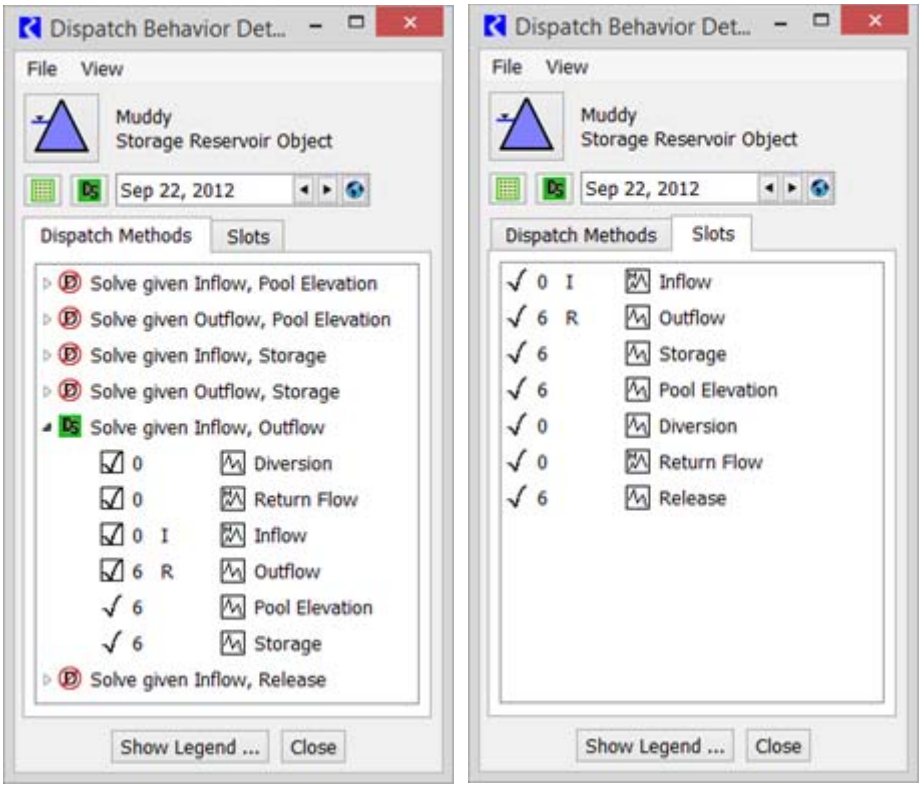
The Dispatch Behavior Details dialog is similar to the one in the Model Run Analysis—Simulation dialog, but contains additional information. In both the Dispatch Methods view and the Slots view, the priorities and flag status of slots is displayed as well as their known/unknown status. The slot priorities and flags, along with knowledge about how objects select a dispatch method, allow the user to understand why each object solved the way it did.

In using the Dispatch Behavior Details dialog to determine why an object solved using a particular dispatch method, it is important to recognize that the display only shows the post-dispatch information. This is different

from the information before the dispatch. After the dispatch, all slots which are solved for have the priority equal to the controller priority at the time of the dispatch.

If, however, the dispatch of an object fails due to a priority conflict, the Dispatch Behavior Details dialog will still show the original priorities and R flags of the slots. This can be helpful in debugging an aborted run. Figure 2.5 illustrates.

Figure 2.5



Comparing Transient and Final Information

An important difference between the diagnostics messages and the Model Run Analysis—Rulebased Simulation dialog is their handling of transient information. Diagnostics messages are generated continuously throughout the run, while the Model Run Analysis—Rulebased Simulation dialog updates its information at the end of each timestep. Diagnostics messages provide a trace of the run execution, including all intermediate states within a timestep. The Model Run Analysis—Rulebased Simulation dialog, however, provides a snapshot of the most recent execution details from the perspective of the end of the timestep. Any intermediate states of dispatching, slot values, and priorities which existed before the solution was finalized for the timestep are unavailable.

Governing Rules

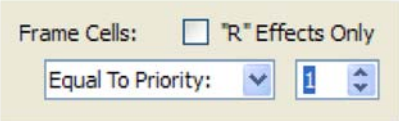
In addition to displaying dispatch information and slot priorities, the Model Run Analysis—Rulebased Simulation dialog allows you to trace the effects of rules across a model. When a rule sets a value which triggers a dispatch, the

dispatch may calculate a new value which will propagate to another object. This object will then redispatch, possibly setting a value which will propagate to yet another object. In this fashion, the effect of a rule may propagate well beyond the object on which it sets a value. Ultimately, it is the highest-priority, successfully executed rules that determine the model solution. The priority of these rules is reflected in the governing slots for each object's last dispatch. The priority of the governing slots identifies the governing rules.

Governing Rules

The governing rules for an object are the rules whose priorities are responsible for the values on the object's governing slots.

In the lower-right corner of the Model Run Analysis—Rulebased Simulation dialog, an area called Frame Cells With Effects: allows the user to show which rules govern the solution. For example, changing the **Option** menu from **None** to **Equal To (==)** and entering **1**, puts a box around all object and timestep pairs that are governed by this priority.



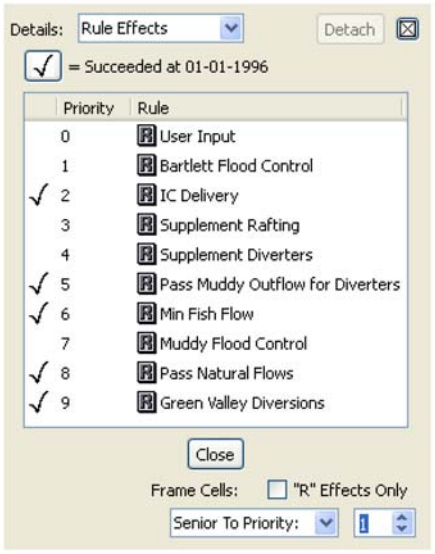
The Effects frame also has a checkbox for enabling **"R" Effects Only**. As the name indicates, selecting this checkbox will only mark the cells of the proper priorities, which were also set directly by a rule.

Rule Effects Details Dialog

The same ability to mark cells which meet priority criteria is also available in the Rule Effects Details dialog. This dialog also displays the names of all rules and whether each rule succeeded at the selected cell's object and time. Finally, this dialog allows you to color-code governing rules for a strong graphical way of tracking their effects across a model. A sample is shown on the following page.

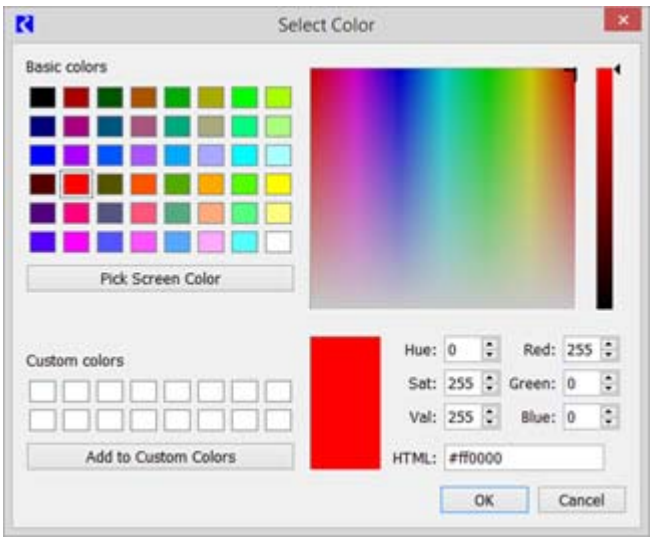
Note: This dialog is always docked within the Model Run Analysis—Rulebased Simulation dialog.

- The Rule Effects dialog is shown by Selecting **View**, then **List Rules / Set Rule Colors** from the Model Run Analysis—Rulebased Simulation dialog or, if a Details dialog is currently docked in that dialog, selecting **Rule Effects** for the details type.

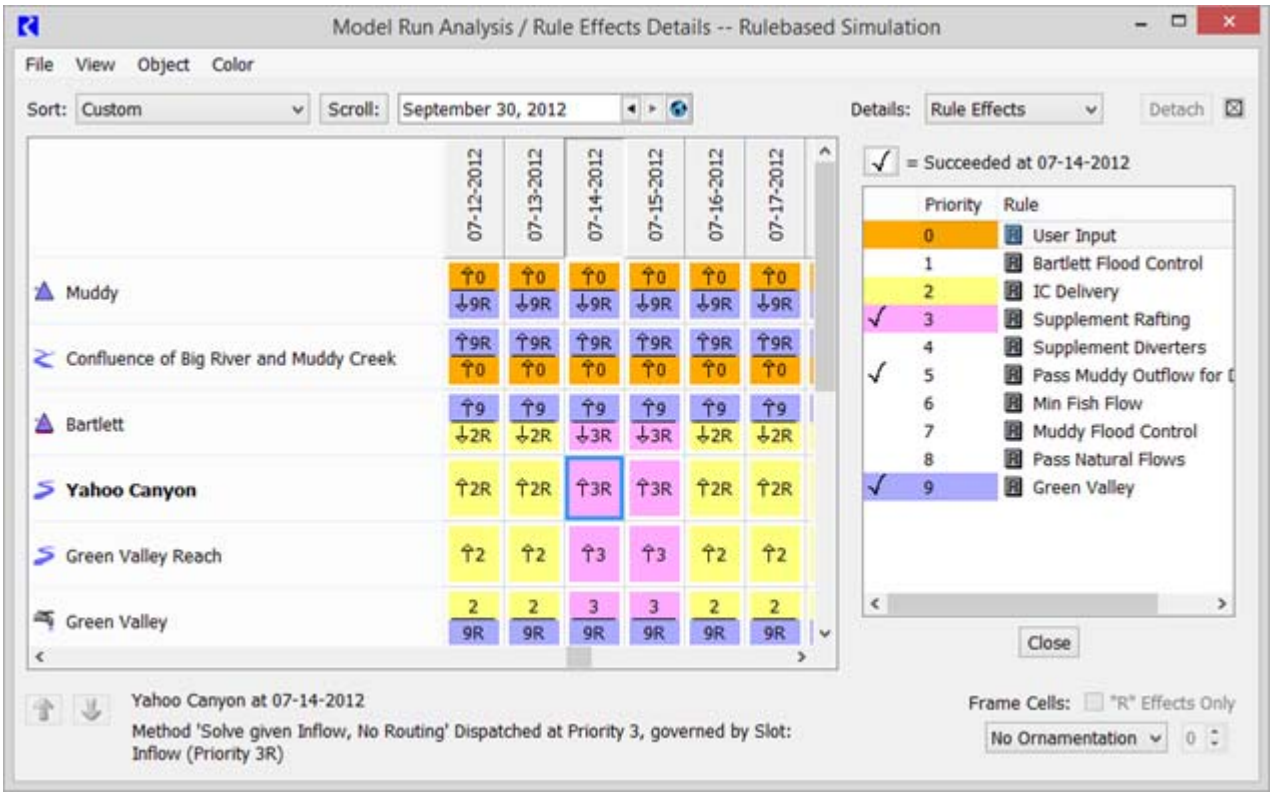


The same commands in the Model Run Analysis—Rulebased Simulation dialog's Effects box are in the bottom of the Rule Effects dialog. Making a selection in this dialog automatically has effect in the other dialog. Governing rules may be selected by name rather than priority number by selecting their name in the top of the dialog. The check marks to the left of each rule indicate whether or not the rule fired successfully on the timestep reported in the middle of the dialog. Selecting each timestep's cell in the Model Run Analysis—Rulebased Simulation dialog lets you see the successful rule firings for that timestep.

To assign colors to governing rules, select a rule in the Rule Effects dialog and select **Color**, then **Edit**. This brings up a color selector for assigning a color to portions of any cells in which this is a governing rule.



Rule Colors can be enabled or disabled by selecting **View**, then **Show Rule Color**.



Chapter 2
Model Run Analysis

Chapter 3
Diagnostics

Topics

- About RiverWare Diagnostics, page 33
- Simulation Diagnostics, page 46
- Rulebased Simulation Diagnostics, page 50
- Optimization Diagnostics, page 59
- Workspace Diagnostics, page 63
- Accounting Diagnostics, page 65

About RiverWare Diagnostics

Diagnostics are a powerful and helpful feature which provide you with the ability to gather useful messages during the course of a run or while building a model. User-selected diagnostic messages are displayed in the Diagnostics Output Window where Error and Warning messages also appear. Diagnostics may be filtered according to their context or turned off altogether. Errors and Warnings may not. Unlike Errors or Warnings, the Diagnostics Output Window will not be forced to the front by diagnostics. It is up to you to open the window.

Diagnostics Process Overview

Following is a quick guide to using diagnostics.

1. Enable diagnostics from the Diagnostics Manager. See"Enable Diagnostics" on page 34, Step 1.
2. Specify the destination of the diagnostics. See "Enable Diagnostics" on page 34, Step 2.
3. Select the diagnostic categories you wish to print. See "Diagnostic Configuration Dialogs" on page 35.
4. Filter the diagnostics by Object, Slot, Timestep, Rule, or Account. See "Context Filtering" on page 37.
5. Apply the changes in both the Diagnostic Configuration dialog and the Diagnostics Manager dialogs.
6. Run the model and view the diagnostics in the specified file or Diagnostics Output. See "Diagnostic Output Window" on page 40.

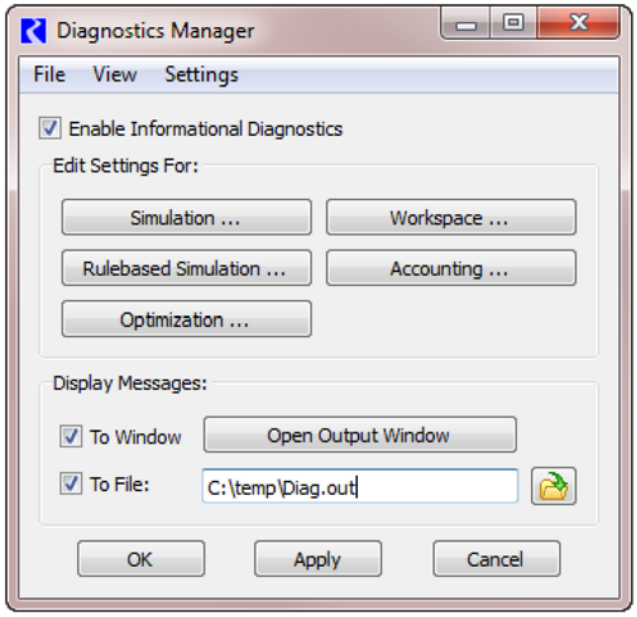
See "RPL Debugging and Analysis" on page 69 for details on debugging RPL logic using diagnostics or other tools.

Diagnostics Manager

The Diagnostics Manager is used to enable informational diagnostics. Informational diagnostics are any messages which, although helpful in debugging, are not indicators of problems which would stop a modeling run. Informational diagnostics should not be confused with RiverWare warnings and errors. Whereas informational diagnostics may be filtered by context and turned off altogether, warnings and errors may not. Warnings and errors are always displayed in the Diagnostics Output window and always force this window to come to the front of other open windows.

The diagnostics Manager can be accessed as follows:

- From the main workspace, select **Utilities**, then **Diagnostics Manager...**
- From the Run Control dialog, select **View**, then **Diagnostics Manager...**
- From the Diagnostics Output Window, select **Settings**, then **Diagnostics Manager...**



RiverWare is initially started with the diagnostic capability disabled primarily for efficiency. Posting, filtering, and outputting unnecessary diagnostic messages may take up a significant portion of the run time. This should be kept in mind when activating diagnostic categories and filters. An excessive amount of diagnostic messages produced at run time may considerably slow down the execution.

Enable Diagnostics

Use the following steps to enable diagnostics.

1. Enable information diagnostics. Select the **Enable Informational Diagnostics** checkbox. Do this only if the checkbox is not already checked. (The diagnostics are enabled when the **Enable Informational Diagnostics** checkbox is selected).

2. Specify diagnostic destination. The destination of output messages is set by selecting the **To Window** and/or **To File** checkboxes. In the case of file output, the file name and path must also be specified in the text field or through the file chooser invoked by selecting **Select File** . The **To Window** option is selected by default.

The Diagnostics Manager dialog also contains three buttons at the bottom: **OK**, **Apply**, and **Cancel**. Changes to the dialog must be followed by selecting either the **Apply** or the **OK** buttons in order for the changes to take effect. The **OK** button closes the dialog in addition to applying any new settings. The **Cancel** button may be pressed to cancel any changes made to the dialog since the last **Apply** or **OK**, and close the dialog.

Diagnostics are divided into categories which appear in the upper portion of the Diagnostics Manager. Each category is a button which opens up the Diagnostics Configuration dialog for that category:

- Simulation Diagnostics, page 46
- Rulebased Simulation Diagnostics, page 50
- Optimization Diagnostics, page 59
- Workspace Diagnostics, page 63
- Accounting Diagnostics, page 65

The first three categories are associated with the controllers of the same name. Accounting diagnostics are enabled only if the Accounting is enabled. For Simulation and Rulebased Simulation, diagnostic messages are generated and filtered *during* a run according to the settings specified for the currently active controller only. The Workspace category, however, is independent of the current controller. It is active *before* and *after* a model run. For example, diagnostics for expression slots that are configured to execute at the end of a timestep must be set up in either the Simulation or Rulebased Simulation category. Diagnostics for expression slots that are evaluated manually must be configured from the Workspace category.

Loading and Saving Diagnostic Configurations

Once you have configured the diagnostics settings, these settings can be saved to a file and loaded at a later time. This allows you to configure and save various combinations of diagnostics configurations and context filters. Settings are loaded and saved by selecting **File**, then **Load Settings** or **File**, then **Save Settings** in the Diagnostics Manager dialog.

Diagnostic Configuration Dialogs

Each of the diagnostics configuration dialogs contains a group of diagnostics to enable or disable, and a set of context filters. The available groups are shown below the title “Show diagnostics for” in the upper left section of the dialog window. Each of these diagnostics groups can be toggled on or off by selecting the checkbox located to the left of the group name. Multiple groups can be turned on at once. Some diagnostics groups are only visible by toggling a tree view triangle.

The mechanism which allows you to narrow down the number of diagnostics that will appear is called context. The various contexts are: object, slot, timestep, and rule. Not all of these contexts are valid for every configurations dialog. For example, generating diagnostics about the dispatching of objects is independent of any Slot context. The Simulation Diagnostics Configuration dialog only allows filtering of messages by Object, Slot, and/or Timestep. Selecting a limited set of a particular context, restricts diagnostic generation to that set. For instance, the

<ALL OBJECTS> value within the **Show O diagnostics for** field means that object-related diagnostics will be generated for every object that exists on the RiverWare workspace. If, however, a limited set of objects is listed within the field, then object-related diagnostics will only be generated for that select set of objects.

For any average size model, enabling this many diagnostics is dangerous. The time spent posting, processing, and displaying diagnostic messages may "freeze" model execution. For a small model, the effect is limited to a noticeable slowdown. In general, diagnostics should never be enabled without considerable context filtering. See "Context Filtering" on page 37 for proper use of context filtering.

Following is a description of the Diagnostics Groups and the Context filtering available in the Diagnostics Manager.

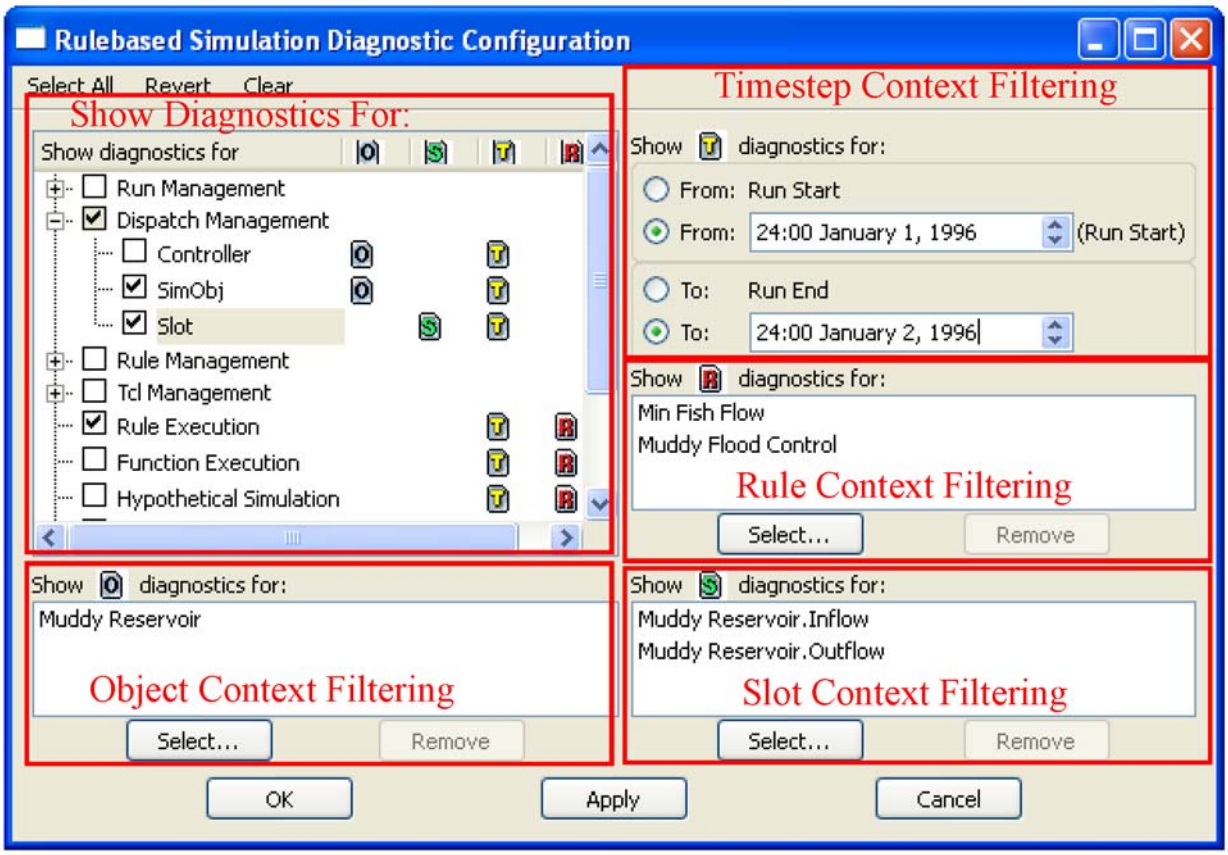
Diagnostics Groups

A diagnostics group is a category of diagnostic messages that pertain to similar events within a run. They include such things as object dispatching information, rules firing information, rules agenda information, slot value information, etc. A complete list of the types of messages issued by each diagnostics group is presented in subsequent sections.

Figure 3.1 shows the Diagnostic Configuration dialog. Diagnostics groups are selected in the upper left portion of the dialog, in the window below the "Show diagnostics for" label. The diagnostics groups are turned on by selecting the checkbox to the left of the group name. When a diagnostics group is enabled, a check mark is shown to the left of the group's name.

Some of the items in the list are actually groups of diagnostics groups. These *super-groups* are purely for organization and generate no diagnostics. The subgroups within a super-group may be viewed by selecting the tree-view plus or arrow symbol to the left of the name. This action will be shown in this manual as **Super-Group**, then **Sub-Group**. When any of the diagnostic subgroups in a super-group are enabled, a check mark is shown to the left of the super-group name, in addition to the check mark to the left of each sub-group name. Any number of diagnostics groups may be enabled at the same time. When you select a Super-Group check mark, it uses a *tri-state*: All On, All off, previous sub-group selection.

Figure 3.1



Context Filtering

Every diagnostic message is generated within a specific context. Contexts are details about the source of a diagnostic message. The contexts for a message may include the Object, Slot, Timestep, and/or Rule within which it originated.

Diagnostic context

Diagnostic context identifies the Object, Slot, Timestep, and/or Rule from within which a message is generated.

All messages which are part of the same diagnostics group will have the same types of contexts. The contexts which are part of each diagnostics group are indicated by a symbol(s) to the right of the diagnostics group name. Following are the icons representing each context type.

- Object context



- Slot context



- Timestep context



- Rule Context



- Account Context



- Goal Context



Each of the icons contains a letter abbreviation of the type of context, **O** for object, **S** for slot, **T** for timestep, **R** for rule, **A** for account, and **G** for optimization goal. Not all Diagnostics Configuration dialogs contain the same contexts. Table 3.1 summarizes the contexts available to all of the Diagnostics Configuration dialogs.

Table 3.1

Diagnostics Configuration Dialog	Available Contexts
Simulation	Object, Slot, Timestep
Rulebased Simulation	Object, Slot, Timestep, Rules
Optimization	Object, Timestep, Goal
Workspace	Object, Slot
Accounting	Object, Slot, Timestep, Account

Exactly how many icons are displayed depends on the Diagnostics Configuration dialog invoked, as indicated in the table above.

Context filters are important for reducing the number of messages produced by a diagnostics group. Context filters allow you to specify the only sources from which a diagnostic message will be displayed. Any messages originating from a different source are ignored.

Most diagnostic groups are affected by context(s). To the right of every diagnostic group listed in the **Show diagnostics for** field, there are zero or more context icons indicating whether or not those particular types of diagnostics are generated within the depicted context types. For example, the posting of diagnostic group **Dispatch Management**, then **Slot** messages is affected by the Slot and Timestep context. Both the Slot and Timestep context icons appear to the right of the listed diagnostic group. In this way, you can determine whether selecting a limited set of items within a context type will affect the frequency of messages of a particular diagnostic group. Choosing a limited set of slot names, for example, only affects the frequency of diagnostics groups depicting the slot context icon **S**.

The summary context icons above the diagnostic group list box indicate whether or not diagnostic groups affected by the specific context are turned on. If none of the diagnostic groups are turned on, then diagnostics are not generated from any of the existing contexts. As a result, each of the summary context icons above the list box is turned OFF, and each has a red line through it. As soon as a single diagnostic group is turned ON and is related to a specific context, the corresponding summary context icon is turned ON (the red line is removed.)

Example 3.1

Enabling the **Dispatch Management,** then **Controller** diagnostics group for a model with 60 objects and 5 years of 1 Month timesteps, would generate at least 3600 messages. This is too many messages to look through to verify that a particular object dispatched on a particular timestep. Because the diagnostics group has both object and time context (as indicated by the symbols to the right of its name), we can set context filters to ignore all messages which did not originate from the desired object and timestep. This reduces the number of diagnostic messages to look through from approximately 3600 to 1.

Object List

The object list in the Simulation Diagnostic Configuration dialog contains a list of the objects for which applicable diagnostics will be generated. To add or delete members of the list, use the **Select** button under the **Show O diagnostics for** field. This displays the Object Selector dialog. Highlighted objects within the list may be removed directly with the **Remove** button.

Timestep Selection

In the upper right corner of the Diagnostics Configuration dialog, there is a **Show T diagnostics for** field, which allows you to input the times within which diagnostics (affected by timestep context) will be generated. By default, diagnostics are posted from the start of the run to the end of the run. This can be overridden by selecting the toggle buttons to the left of the **From** and **To** fields. In addition, there is a toggle to specify the Run Start and Run End. The date/time spinners to the right of text fields can also be used to select the desired times.

Slot List

The Slots list is modified in the same manner as the Objects list.

Rule List

In the Rulebased Simulation Diagnostic Configuration dialog, a box is displayed to allow you to select rules for which diagnostics should be displayed. See "Selecting Rules" on page 56 for details.

Accounts List:

In the Accounting Diagnostics Configuration dialog, a box is displayed to allow you to select accounts for which diagnostics should be displayed.

Goal List

In the Optimization Diagnostic Configuration dialog, a box is displayed to allow you to select goals for which diagnostics should be displayed.

Select All, Revert, and Clear

These three menu options are available within each of the diagnostics configuration dialogs, depending on whether or not they exist in the specific Configuration dialog. Each of these options can be applied to diagnostics groups, the object list, the slot list, the rule list, and the timestep text fields.

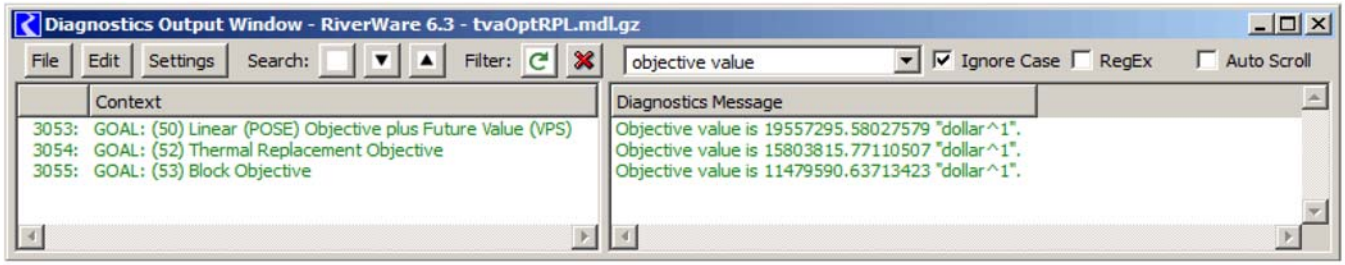
- Applying the **Select All** feature has the following effects:
  - Diagnostics Groups: All turned to ON

  - Objects: ALL listed in listbox
  - Slots: ALL listed in listbox
  - Rules: ALL listed in listbox
  - Goals: ALL listed in listbox
  - Times: Set to StartTime and EndTime

- The **Revert** feature reverts the appropriate items back to their state prior to your modifications. Every time the **Apply** button is pressed, your changes are recorded. Selecting the **Revert** command will revert back to the settings which were last recorded.
- Applying the **Clear** feature has the following effects:
  - Diagnostics Groups: All turned to OFF
  - Objects: NONE listed in listbox
  - Slots: NONE listed in listbox

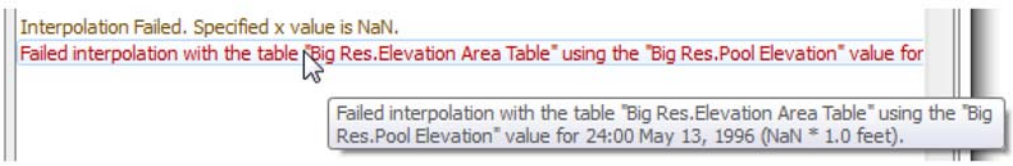
Diagnostic Output Window

The Diagnostics Output Window displays the diagnostics during a RiverWare run. Error and Warning messages are also displayed in this window, regardless of user settings, and automatically force the window to the top of any other windows. The object and timestep for which a message is generated are listed under the **Context** bar. The message containing the diagnostic information is listed under the **Diagnostic Message** bar on the same line. When the **Auto Scroll** checkbox is selected, the window will continually scroll down as messages are output. In a large model where many diagnostics are generated, the continuous scrolling can add significantly to the run time. In these cases, **Auto Scroll** can be disabled by clearing the checkbox.

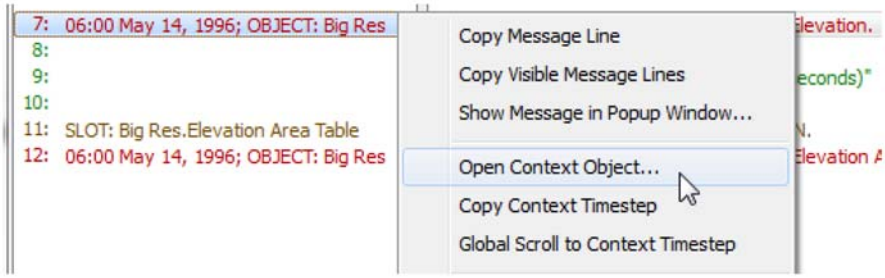


You can scroll vertically and horizontally in both the context and message fields. In addition to scrolling in the fields, you may wish to resize the entire Diagnostics Output Window. This is useful when many long messages and/or contexts need to be read. To move the divider between the fields, position the mouse over the space between the heading bars, press the left mouse button, and drag the divider to the desired width.

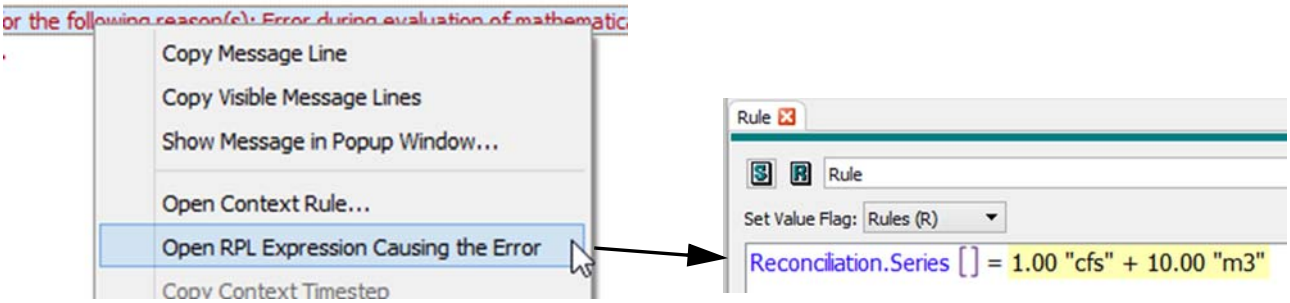
Long diagnostic messages are shown as a wrapped tooltip when you hover over a message.



- In the Diagnostics Output Window, the following right-click context menu operations are available:

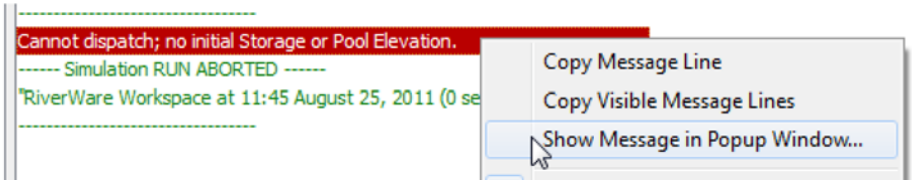


- **Open Context <Item>**. Open the dialog for the listed item:
  - Object,
  - Account,
  - Slot,
  - RPL Set,
  - RPL Rule,
  - RPL Goal, or
  - RPL Group
- **Open RPL Expression Causing the Error.** When an error occurs in RPL evaluation, this menu becomes available. When selected, the appropriate RPL dialog will be opened and the expression causing the error will be selected.

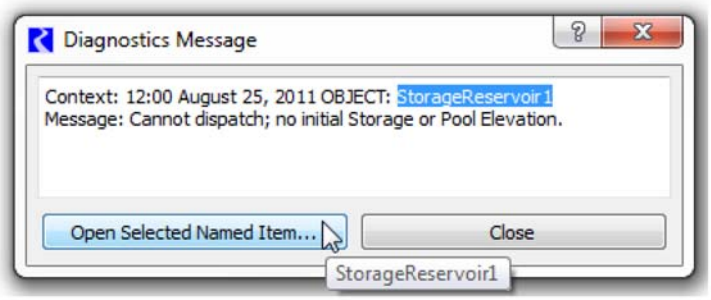


- **Copy Context Timestep**. Copy the datetime to the clipboard
- **Global Scroll to Context Timestep**. Global scroll to the datetime

If you wish to read a diagnostic in a wrapped window, right-click the line of text and choose Show Message in Popup Window.



This opens a dialog that shows the full diagnostics text, including the context, in a wrapped window. If you highlight the name of an Object, Slot, Account, RPL item (Set, Group, Rule, Function), you can select the **Open Selected Named Item** menu to open the corresponding dialog.

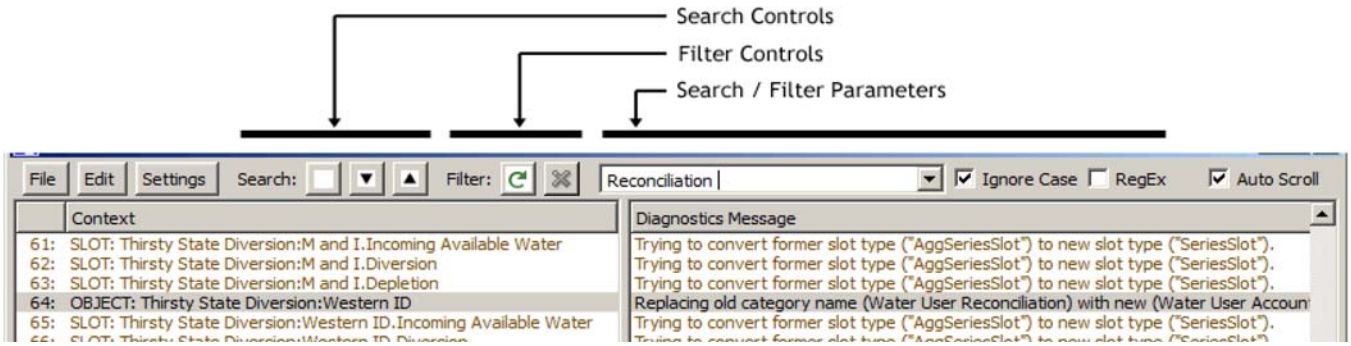


The same right-click context menus listed in the bullets above are available in this dialog too. The keyboard accelerators Ctrl-A for Select All and Ctrl-C for Copy are also available on the text in the dialog.

Message Search and Filter Controls

There are two approaches used to find messages: **Search** and **Filter**. Search to look for a specific text string row by row. Filter to show all rows that match the text string. (You can also search on filtered results) The two approaches share the text entry field. Following are descriptions of the controls and options:

Figure 3.2

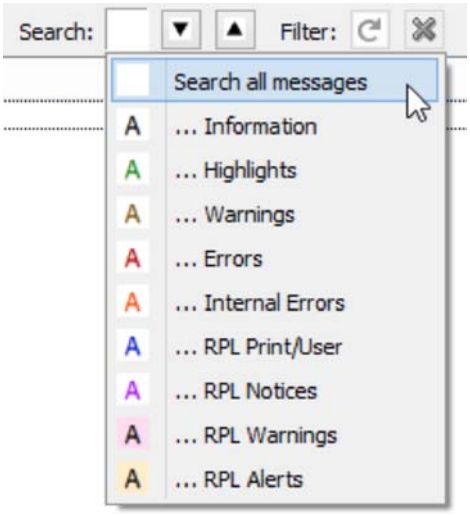


Search Controls

The **Search** controls include: the following:

- **Message Type** (color coded) menu. Filter the search by the type of message displayed, i.e. by color. Select the white box just to the right of the **Search** field. This opens a menu (see Figure 3.3). Choose an item and any subsequent searches will be filtered to only include messages of that type. If you just want to see the next message of a certain type, don't enter anything in the search box; a search will jump to the next message of the indicated color/type.

Figure 3.3 Diagnostics search message type filter menu



- **Search Forward** button (down) 
- **Search Backward** button (up) 

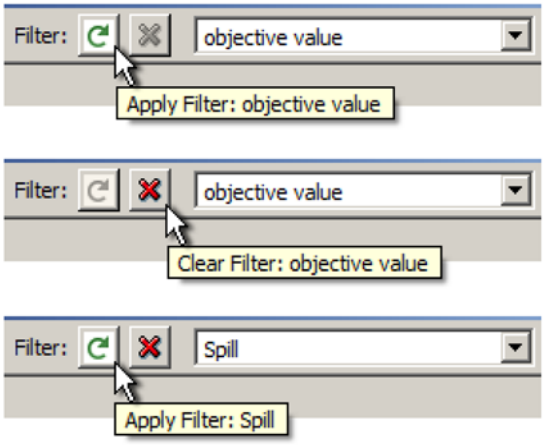
  Note: The **Add entered text as User Message** selection in the **Edit** menu can be used to effectively append a bookmark to the end of the message list. That text can later be recalled from the history menu of the entry field, and used to scroll to the various instances of that bookmark within the message list using the search functions.

Filter Controls

Filtering allows you to show only those lines that match the specified text. The **Filter** controls include the following buttons:

- The **Apply Filter** button  applies the entered text as the active filter pattern. Message rows not matching the pattern are hidden. This button is disabled when the search / filter parameters already match the currently active filter (or when the entered text is empty).

Figure 3.4



- The **Clear Filter**  button clears filtering -- causing all message rows which had been hidden to become visible again. This button is enabled only when filtering is active.

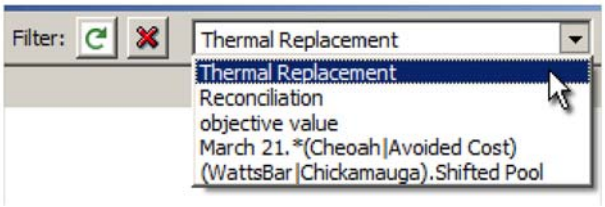
The only direct indication that messages are being filtered is the Clear Filter (red "X") button being enabled and discontinuous line numbers in the message list.

When applying or clearing a filter, the view is re-scrolled to the currently selected item (when that item remains non-hidden). So, it is possible to use filtering to locate a diagnostic message (and select that message), and then view that message in the context of all generated messages by turning filtering off.

Search and Filter Parameters

Enter your desired search/filter text pattern in the text box. This is an editable "combo box" to support a history of recently applied patterns, as shown in Figure 3.5.

Figure 3.5



When typing in the entry field, hitting Enter is equivalent to selecting **Search Forward**, and pressing **Shift-Enter** is equivalent to selecting **Search Backward**.

The following options apply to both the Search and Filter functions.

- The **Ignore Case** checkbox causes upper/lower case differenced to be ignored when comparing the pattern text to message text.
- The **RegEx** checkbox changes the interpretation of the pattern text from "Wild Card" to "Regular Expression" syntax. (See the following section).

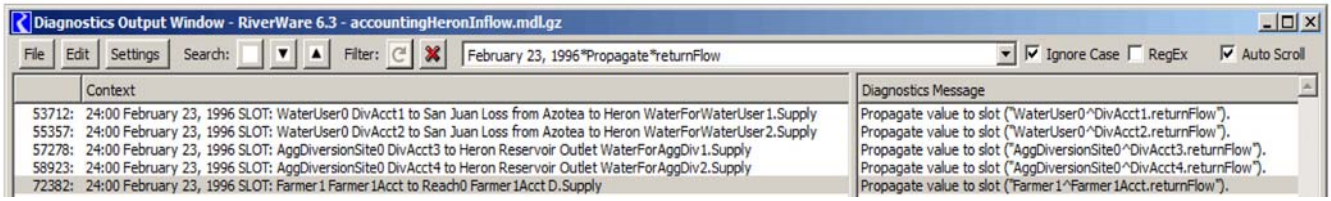
Note: The search functions skip over messages which have been hidden by filtering.

Search and Filter Pattern Matching Technical Details

A given message line matches the pattern if the line text contains the pattern. (This is different from the pattern matching the entire line of text). Application of the pattern match concatenates the "context" and "message" parts of the message row, rather than independently looking into those two parts. This makes possible, for example, a wildcarded expression which matches pieces from both the context and message parts of the message line, as follows:

```
February 23, 1996*Propagate*returnFlow
```

Figure 3.6



The following is the definition for *wild card* pattern matching (i.e. not RegEx):

- Any character represents itself apart from those mentioned below.
- "?" (question mark) matches any single character. It is the same as "." (period) in regular expressions.
- "*" (asterisk) matches zero or more of any characters. It is the same as ".*" (period asterisk) in regular expressions.
- Sets of characters can be represented in square brackets [...].
- A back-slash ('\') "escapes" the wild card characters. It is possible to match a question mark, asterisk, or square bracket by preceding ("escaping") those characters with a back-slash ("\").

Regular Expression ("RegEx", "regex" or "regexp") pattern syntax is an advanced, potentially complex pattern matching format developed originally with Unix-based tools. It's not described further in this document but is widely referenced online.

File Menu

The **File** menu includes the following selections:

- **Save Diagnostic Messages to File.** Save all diagnostics in the window to a text file
- **Show Diagnostics Messages in Stand Alone-Window**. When the diagnostics output is docked in an SCT, this menu option allows you to show the messages in a stand-alone window
- **Show Workspace**. bring the workspace to the front
- **Close Diagnostics**: Close the Diagnostics Output Window

Edit Menu

The **Edit** menu the following selections:

- **Copy Message Line**. Copy a single line to the clipboard
- **Copy Visible Message Lines**. Copy visible lines to the clipboard
- **Copy All Message Lines**. Copy all the diagnostics to the clipboard
- **Add entered text as User Message**. This option takes any text that is in the Search text box and creates a new message line. This is useful in debugging if you want to place a marker
- **Clear Messages**. This command clears the existing diagnostics from the scrollable listbox. During the course of running RiverWare with diagnostics enabled, the number of diagnostics can grow very large (especially if they are being generated for **<ALL OBJECTS>** and **<ALL SLOTS>** context types, and all Times). As a result, **Clear Messages** whenever possible to decrease the amount of memory used by RiverWare. Diagnostics are appended to the bottom of the window and are not deleted from the window until you invoke the **Clear Messages** command. As a result, if you are making multiple runs, the diagnostics from the first run remain visible at the top of the window. When looking at diagnostics, make sure to always start at the bottom to ensure you are viewing diagnostics from the current run.

Settings Menu

The **Settings** menu includes the following selections:

- **Diagnostics Manager.** Open the diagnostics manager.
- **Show Color Legend.** Display the meaning of diagnostic message colors.
- **Show Column Headers.** Toggle to either show or hide the column headings.
- **Show Line Numbers.** Toggle to either show or hide line numbers.
- **Show Filter Statistics.** Toggle to either show or hide a statistics panel below the message list. This indicates the total number of diagnostics messages, and how many are shown and hidden due to message filtering.
- **Show Horizontal Scroll Bars.** Toggle to either show or hide scroll bars
- **Search/Filter: Ignore Case**. Toggle to ignore case in the search/filter box.
- **Search/Filter: Use Regular Expressions.** Toggle to use Regular Expressions in the search/filter box.
- **Automatic Width Adjustment**. The Context portion of the message list (on the left side) is given the full width necessarily to display its content for all non-hidden messages.
- **Extra Width Fix.** This option is intended to correctly measure the text content on certain Windows machines. When this option is on, automatically adjusted widths are increased by about 12%

Simulation Diagnostics

Figure 3.7 is a sample Simulation Diagnostic Configuration Dialog. Available filters include Object, Timestep, and Slots.

Figure 3.7

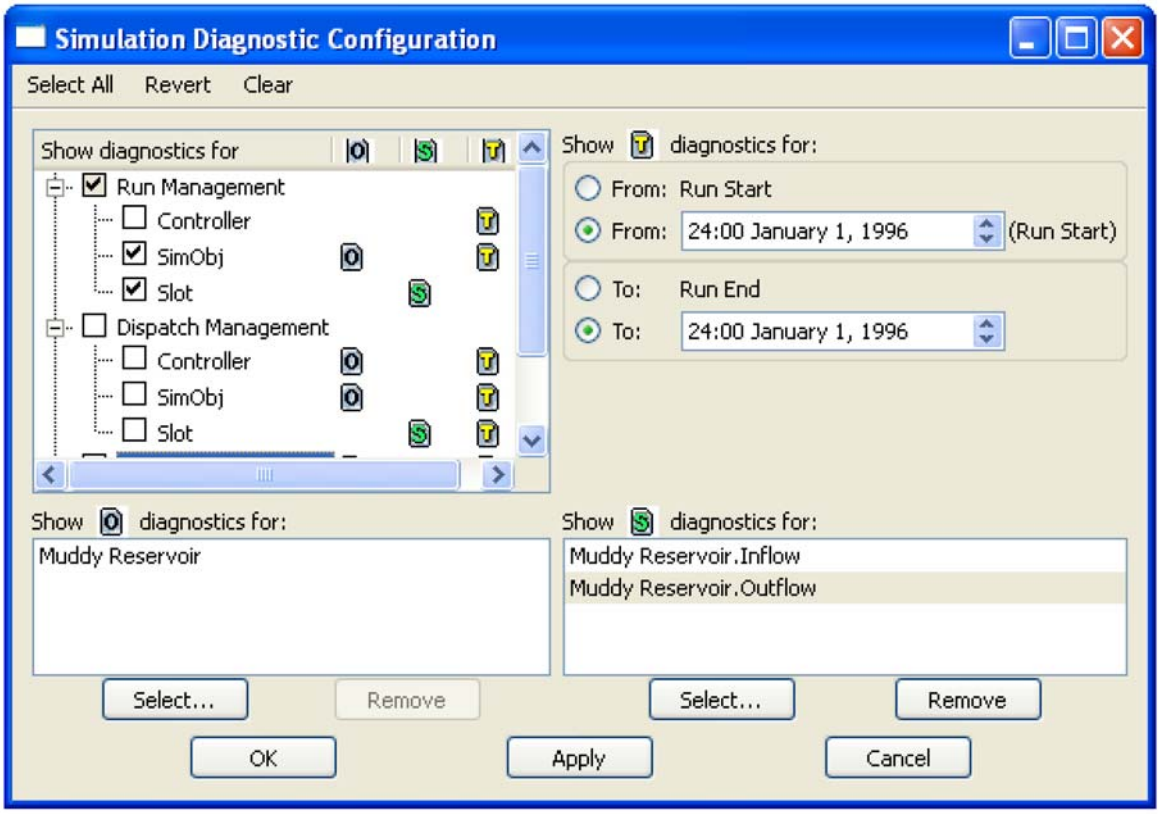


Table 3.2 summarizes the types of messages that are displayed when a Diagnostic Group is selected.

Table 3.2

Super Group	Diagnostics Group	Type of diagnostics issued
Run Management:		
	Controller	Beginning/End of Run/Timestep for each time. **Examples:** • Initializing model run. • Beginning of Run. • Begin Timestep. • Execute timestep. • End of run.

Table 3.2 (Continued)

Super Group	Diagnostics Group	Type of diagnostics issued
	SimObj	Beginning/End of Run/Timestep for each Object and time. **Examples:** • Clear state. • Propagating user input. • Beginning of run. • Begin timestep. • End timestep. • End of run.
	Slot	Beginning of Run for each Slot and time. **Examples:** • Clear state. • Propagating user input.
Dispatch Management:		
	Controller	Beginning of a dispatch for each object and time. **Example:** • Executing dispatch method (Solve given Inflow, Outflow) at Priority 7.
	SimObj	Dispatch readiness for each object and time. **Examples:** • Slot (“Inflow”) added to known set. • Ready for dispatch method (“Solve given Inflow, No Routing”). • Slot (“Inflow”) is already in known set. • Re-dispatching using slot priorities. Priority 3R:Inflow Priority 7:Outflow Priority 7:Total Diversion Priority 7:Total Return Flow Priority 7:Total Local Inflow Conflicting slot values:(“Inflow” “Outflow” “Total Diversion” “Total Return Flow” “Total Local Inflow”).
	Slot	Value assignments and propagation for each slot and time. **Examples:** • Set value = 453.003351 *1000.0 acre-feet/month = 208.621343 *1.0cms. • Propagate value to slot (“Total Available Water”).
Independent Groups:		
	User Methods	Execution of User Methods for each object and time. **Example:** • Executing user method (“DailyEvaporationCalc”).

Table 3.2 (Continued)

Super Group	Diagnostics Group	Type of diagnostics issued
	Interpolation	TableSlot table interpolation lookups for each slot. **Examples:** • Table Interpolation 2 (3D) Specified z value (960.6579) falls between row ranges (556-572) and (573-589). (1->2) rows (556-572) x=195.804788, y=198.503173 (1->2) rows (573-589) x=195.804788, y=198.585469 (fixed 0 at 960.657920, 1->2) (in=195.804788) (out=198.579323 slope=.656000 intercept=70.131382)
	Client Server	Provides information on the Client Server interaction. This includes the RiverWare - DSS, RiverWare - HDB, and RiverWare - MODFLOW connections.
	Expr. Slot Execution	Expression slot execution are printed for expression slots that are configured to evaluate during a run (evaluation time of beginning or end of timestep or beginning or end of run). **Example:** • Expression evaluated to 624.00 [1000 acre-ft]
	Expr. Slot Function Execution	Evaluation of RPL functions called by expression slots evaluated during a run. Diagnostics messages are printed for function execution (both user defined and predefined). “Before Execution” function diagnostics show the function being called and the arguments passed into the function. “After Execution” function diagnostics show the value that the function evaluated to and whether or not the value was constrained. **Note:** Function diagnostics also need to be enabled on the functions themselves. This can be done through the Function Editor dialog (for enabling diagnostics on individual functions) or by selecting **Set**, then **Function Diagnostics** from the main **Set Editor** (for enabling or disabling function diagnostics for all functions in the set). The Expression Slot Set editor is accessed from the Workspace **Policy**, then **Open Expression Slot RPL Set**. See “Function Diagnostics” on page 57 for details on configuring the diagnostics. See “Series Slots With Expression” in *User Interface* for details on expression slots. **Example:** • SumFlowsToVolume($”Mountain Storage.Inflow”, @”Start Timestep”, @”t”) • “SumFlowsToVolume” evaluated to: 274839.74 [“m3”].
	Init. Rules Rule Execution	Initialization Rules rule execution diagnostics are printed for each rule. For a Simulation run, there is no filtering based on rule name. When enabled, diagnostics are shown for all Initialization rules. **Examples:** • Executing rule (#3) Assignment initiated (the left-hand side is “Mohave.Storage”[]) Evaluation of Assignment statement successful; will attempt assignment Mohave.Storage[July, 1998] = 1590800.00 [1.00 acre-ft] Rule successfully finished • Early termination: NaN encountered. The problem was encountered at the following location within the expression: “FC Violation”(). • RHS of assignment evaluated to NaN.

Table 3.2 (Continued)

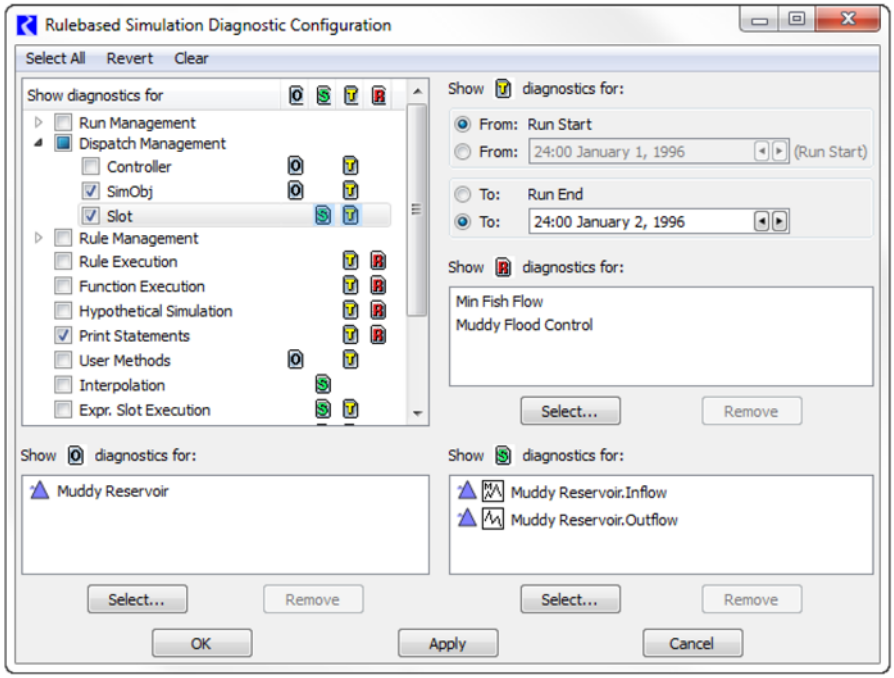
Super Group	Diagnostics Group	Type of diagnostics issued
	Init. Rules Function Execution	Diagnostics messages for function execution (both user defined and predefined) from initialization rules. For a Simulation run, there is no filtering based on rule name. When enabled, diagnostics are shown for all Initialization rules. "Before Execution" function diagnostics show the function being called and the arguments passed into the function. "After Execution" function diagnostics show the value that the function evaluated to and whether or not the value was constrained. **Note:** Function diagnostics also need to be enabled on the functions themselves. This can be done through the Function Editor dialog (for enabling diagnostics on individual functions) or by selecting **Ruleset**, then **Ruleset Function Diagnostics** from the main **Ruleset Editor** (for enabling or disabling function diagnostics for all functions in the ruleset). See "Function Diagnostics" on page 57 for details. **Examples:** • GetMaxReleaseGivenInflow (% "WattsBar", 0.634 ["1000 * cms"], @"Current Timestep") • "GetMaxReleaseGivenInflow" evaluated to: 1299.413 ["cms"] • "GetMaxReleaseGivenInflow" constrained to: 1200.00 ["cms"]
	Init. Rules Print Statements	User defined messages for each initialization rule. For a Simulation run, there is no filtering based on rule name. When enabled, diagnostics are shown for all Initialization rules. **Example:** • Entering rule curve routine for January - July

Rulebased Simulation Diagnostics

The importance of diagnostics in debugging Rulebased Simulation models cannot be overstated. Diagnostics allow the user to examine rule firing, rule success/failure, slot assignments, and dispatching in the chronological order in which these events take place. It is the only way to verify that the intended interaction between rules and object dispatching is being carried out exactly as expected. Becoming familiar with configuring and reading diagnostics will save you hours and hours of speculation when your model does not behave the way you planned. See "RPL Debugging and Analysis" on page 69 for details on debugging Rules and other RPL logic using diagnostics. See "Types of RPL Debugging and Analysis" on page 69 for a general discussion on debugging RPL logic.

The Rulebased Simulation Diagnostics Configuration dialog allows you to select from fifteen diagnostics groups and four context filters which are relevant during a Rulebased Simulation run. This dialog's settings are only valid when the Rulebased Simulation controller is selected. It is accessed from the Rulebased Simulation button on the Diagnostics Manager.

Figure 3.8



Rulebased Simulation Diagnostic Groups

Table 3.3 summarizes the types of messages that are displayed when a Diagnostic Group is selected.

Table 3.3

Super Group	Diagnostics Group	Type of diagnostics issued
Run Management:		
	Controller	Beginning/End of Run/Timestep for each time. **Examples:** • Initializing model run. • Beginning of Run. • Begin Timestep. • Execute timestep. • End of run.

Table 3.3 (Continued)

Super Group	Diagnostics Group	Type of diagnostics issued
	SimObj	Beginning/End of Run/Timestep for each Object and time. **Examples:** • Clear state. • Propagating user input. • Beginning of run. • Begin timestep. • End timestep. • End of run.
	Slot	Beginning of Run for each Slot and time. **Examples:** • Clear state. • Propagating user input.
Dispatch Management:		
	Controller	Beginning of a dispatch for each object and time. **Example:** • Executing dispatch method (Solve given Inflow Outflow) at Priority 7.
	SimObj	Dispatch readiness for each object and time. **Examples:** • Slot (“Inflow”) added to known set. • Ready for dispatch method (“Solve given Inflow, No Routing”). • Slot (“Inflow”) is already in known set. • Re-dispatching using slot priorities. Priority 3R:Inflow Priority 7:Outflow Priority 7:Total Diversion Priority 7:Total Return Flow Priority 7:Total Local Inflow Conflicting slot values:(“Inflow” “Outflow” “Total Diversion” “Total Return Flow” “Total Local Inflow”).
	Slot	Value assignments and propagation for each slot and time. **Examples:** • Set value =453.003351 *1000.0 acre-feet/month =208.621343 *1.0cms. • Propagate value to slot (“Total Available Water”).
Rule Management:		
	Dependencies	Dependencies for each rule and time. **Example:** • After execution, rule #4 has the following dependencies: Mead.Storage, Mead.Inflow.

Table 3.3 (Continued)

Super Group	Diagnostics Group	Type of diagnostics issued
	Agenda	Additions and status of the agenda for each rule and time. **Examples:** • The following rules are currently on the agenda: 3, 4, 8. • Changing dependency (Mead.Outflow) caused rule #3 to be put on the agenda.
	Assignments	Value assignments for each rule, time and slot. **Example:** • Result of sending values to controller: - Priorities permitted setting of Mead.Storage[24:00 January 31, 1993] to 676194743.34m3
Independent Groups:		
	Rule Execution	Rule execution details for each rule, time and slot. **Examples:** • Executing rule (#3) Assignment initiated (the left-hand side is “Mohave.Storage”[]) Evaluation of Assignment statement successful; will attempt assignment Mohave.Storage[July, 1998] = 1590800.00 [1.00 acre-ft] Rule successfully finished • Early termination: NaN encountered. The problem was encountered at the following location within the expression: “FC Space Violation”(). • RHS of assignment evaluated to NaN. • Unable to set slot (Outflow at 00:00 February 1, 1999) Rule’s priority (4) junior to slot’s (2).
	Function Execution	Diagnostics messages for function execution (both user defined and predefined). “Before Execution” function diagnostics show the function being called and the arguments passed into the function. “After Execution” function diagnostics show the value that the function evaluated to and whether or not the value was constrained. **Note:** Function diagnostics also need to be enabled on the functions themselves. This can be done through the Function Editor dialog (for enabling diagnostics on individual functions) or by selecting **Ruleset**, then **Ruleset Function Diagnostics** from the main **Ruleset Editor** (for enabling or disabling function diagnostics for all functions in the ruleset). See “Function Diagnostics” on page 57 for details. **Examples:** • GetMaxReleaseGivenInflow (% “WattsBar”, 0.634 [“1000 * cms”], @”Current Timestep”) • “GetMaxReleaseGivenInflow” evaluated to: 1299.413 [“cms”] • “GetMaxReleaseGivenInflow” constrained to: 1200.00 [“cms”]

Table 3.3 (Continued)

Super Group	Diagnostics Group	Type of diagnostics issued
	Hypothetical Simulation	Diagnostics messages for hypothetical simulation. This includes information about the start and end of hypothetical simulation, the iteration information (if using a target hypothetical simulation function), and any information regarding the hypothetical simulation solution. **Examples:** • HypSim: Hypothetical limit simulation starting for specified SubBasin. • Hypothetical simulation occurring within the time range [12:00 November 23, 1995 - 00:00 December 5, 1995]. • HypSim: For iteration 1, control slot interval is [0.000000 “cms”, 75.000000 “cms”], range of interval is [1768.684936 “m”, 1763.270439 “m”]. • HypSim: For target value 1768.297200 “m”, result is control value 19.628906 “cms”, leading to the simulated target value 1768.296428 “m” (error = 0.000772 “m”). • HypSim: Hypothetical limit simulation ending for specified SubBasin.
	Print Statements	User defined messages within rules for each rule and time. **Example:** • Entering rule curve routine for January - July
	User Methods	Execution of User Methods for each object and time. **Example:** • Executing user method (“DailyEvaporationCalc”).
	Interpolation	TableSlot table interpolation lookups for each slot. **Examples:** • Table Interpolation 2 (3D) Specified z value (960.6579) falls between row ranges (556-572) and (573-589). (1->2) rows (556-572) x=195.804788, y=198.503173 (1->2) rows (573-589) x=195.804788, y=198.585469 (fixed 0 at 960.657920, 1->2) (in=195.804788) (out=198.579323 slope=.656000 intercept=70.131382)
	Expr. Slot Execution	Expression slot execution diagnostics are printed for expression slots that are configured to evaluate during a run (evaluation time of beginning or end of timestep, or beginning or end of run). **Example:** • Expression evaluated to 624.00 [1000 acre-ft]

Table 3.3 (Continued)

Super Group	Diagnostics Group	Type of diagnostics issued
	Expr. Slot Function Execution	Evaluation of RPL functions called by expression slots evaluated during a run. Diagnostics messages are printed for function execution (both user defined and pre-defined). “Before Execution” function diagnostics show the function being called and the arguments passed into the function. “After Execution” function diagnostics show the value that the function evaluated to and whether or not the value was constrained. **Note**: Function diagnostics also need to be enabled on the functions themselves. This can be done through the Function Editor dialog (for enabling diagnostics on individual functions) or by selecting **Set**, then **Function Diagnostics** from the main **Set Editor** (for enabling or disabling function diagnostics for all functions in the set). The Expression Slot Set editor is accessed from the Workspace **Policy**, then **Open Expression Slot RPL Set**. See “Function Diagnostics” on page 57 for details on configuring the diagnostics. See “Series Slots With Expression” in *User Interface* for details on expression slots. **Example:** • SumFlowsToVolume($”Mountain Storage.Inflow”, @”Start Timestep”, @”t”) • “SumFlowsToVolume” evaluated to: 274839.74 [“m3”].
	Init. Rules Rule Execution	Initialization Rules rule execution diagnostics are printed for each rule. **Examples:** • Executing rule (#3) Assignment initiated (the left-hand side is “Mohave.Storage”[]) Evaluation of Assignment statement successful; will attempt assignment Mohave.Storage[July, 1998] = 1590800.00 [1.00 acre-ft] Rule successfully finished • Early termination: NaN encountered. The problem was encountered at the following location within the expression: “FC Violation”(). • RHS of assignment evaluated to NaN.
	Init. Rules Function Execution	Diagnostics messages for function execution (both user defined and pre-defined) from initialization rules. “Before Execution” function diagnostics show the function being called and the arguments passed into the function. “After Execution” function diagnostics show the value that the function evaluated to and whether or not the value was constrained. **Note:** Function diagnostics also need to be enabled on the functions themselves. This can be done through the **Function Editor** dialog (for enabling diagnostics on individual functions) or by selecting **Ruleset, then Ruleset Function Diagnostics** from the main **Ruleset Editor** (for enabling or disabling function diagnostics for all functions in the ruleset). See “Function Diagnostics” on page 57 for details. **Examples:** • GetMaxReleaseGivenInflow (% “WattsBar”, 0.634 [“1000 * cms”], @”Current Timestep”) • “GetMaxReleaseGivenInflow” evaluated to: 1299.413 [“cms”] • “GetMaxReleaseGivenInflow” constrained to: 1200.00 [“cms”]

Table 3.3 (Continued)

Super Group	Diagnostics Group	Type of diagnostics issued
	Init. Rules Print Statements	User defined messages for each initialization rule. **Example:** • Entering rule curve routine for January - July
Data Management	Rule DMI	Execution of rule DMI's for each specified rule.

Selecting Rules

In the **Rulebased Simulation Diagnostics Configuration**, you are able to select rules for which you wish to see diagnostics. This is done by selecting the **Select** button from the **Show Rules Diagnostics for:** section. If you have both a RBS ruleset and a non-empty Initialization ruleset, you must choose from which set's rules you would like to choose. The Rule Selector then opens. See "RPL Printing and Formatting" in *RiverWare Policy Language (RPL)* for details on using this dialog.

Also, all rules can be added from the **Rulebased Simulation Diagnostics Configuration**, **Select All, then Rules** menu. Selecting **OK** confirms and closes the dialog or selecting **Cancel**, cancel the selection.

Posting User Messages to Diagnostics

There are six ways to post a user-generated message to the Diagnostics Output Window from within RPL:

- Print Statements
- Notice, Warning, and Alert Statements
- Stop Run Statements
- Stop Run Expressions

For all six of these statements or expression types, you can define the message that is displayed in the diagnostics output. The unspecified <expr> of the statement or expression can be any concatenation of the different RPL expression types. All of the elements in the expression are interpreted as strings and formatted into the message.

Print Statements

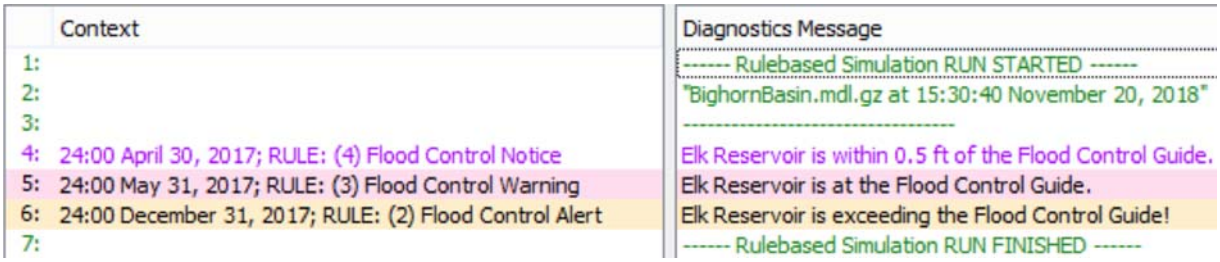
A Print statement can be added to any rule or goal as a top level statement. From the **Statement** menu in the rule or goal, select **Add Print**. In order for the message to appear in the Diagnostics Output Window, you must enable Informational Diagnostics in the Diagnostics Manager, and you must enable the Print Statements diagnostics group in the Rulebased Simulation or Optimization Diagnostics Configuration and select the appropriate rules or goals for which to display diagnostics. Print statements are shown in the Diagnostics Output Window as blue text.

Notice, Warning, and Alert Statements

Notice , Warning, and Alert statements have the same behavior, but they post messages with different colors; see Figure 3.9. The three statement types allow you to post messages with different levels of severity. A Notice, Warning, or Alert statement can be added to any rule or goal as a top level statement. From the **Statement** menu in the rule or goal, select **Add Notice/Warning/Alert**. Unlike Print statements, Notice, Warning, and Alert statements are always shown in the Diagnostic Output Window, regardless of diagnostics settings.

- Notices are displayed as purple text with a white background.
- Warnings are displayed as black text with a pink background.
- Alerts are displayed as black text with an orange background.

Figure 3.9 Examples of Notice, Warning and Alert messages shown in diagnostics output



Stop Run Statements

A Stop Run statement can be added to any rule or goal as a top level statement. From the **Statement** menu in the rule or goal, select **Add Stop Run**. The statement will cause the run to abort, post the specified message with red text, and bring the Diagnostics Output Window to the front. Typically a Stop Run statement will be inside of an If statement so that it only stops the run under specific conditions.

Stop Run Expressions

A Stop Run expression has the same behavior as a Stop Run statement, but it is selected off of the RPL Palette as an expression within the RPL logic rather than at the statement level. Because it is added at the expression level, a Stop Run expression can be added on the right-hand-side of a rule assignment, within a function, or within any other RPL expression. The Stop Run expression will cause the run to abort, post the specified message with red text, and bring the Diagnostics Output Window to the front. Typically a Stop Run expression will be inside of an If/Else expression so that it only stops the run under specific conditions.

Function Diagnostics

To understand more about a rulebased simulation, it is sometimes useful to know more about function execution. RiverWare provides two types of diagnostic messages related to function execution:

- Before Execution diagnostic: The diagnostic is issued just before a function is executed and gives the name of the function and the values passed in as arguments.
- After Execution diagnostic: This diagnostic is issued after the function has completed execution and gives the name of the function and the value to which it evaluated.

To receive diagnostics for a function, you must do the following:

- Open the Diagnostic Manager (accessible, for example, through the Run Control View menu), enable informational diagnostics and select whether they should go to the Output Window or a file.

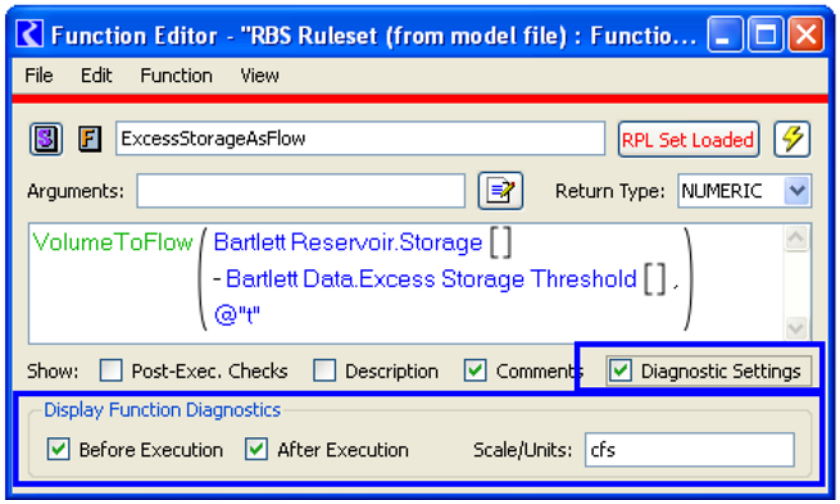
  Note: This step is required to receive any informational diagnostics.

- Open the **Rulebased Simulation Diagnostics Configuration** window and toggle on the Function Execution category.

  Note: This category can be filtered by timestep and rule, if desired.

- Open the Function Editor for the function for which you would like to see diagnostics. Select **View, then Show Diagnostic Settings** or select the **Diagnostic Settings** toggle. This will add a panel to the Function Editor which allows you to toggle on or off either type of diagnostic (**Before Execution** or **After Execution**) for that particular function (see graphic). If the return type of the function is NUMERIC and you would like to see **After Execution** diagnostics for that function, then you may enter the scale and units in which you would like the function's return value to appear within the diagnostic. Examples include 1000 cfs and ft. No quotes are required.

Figure 3.10



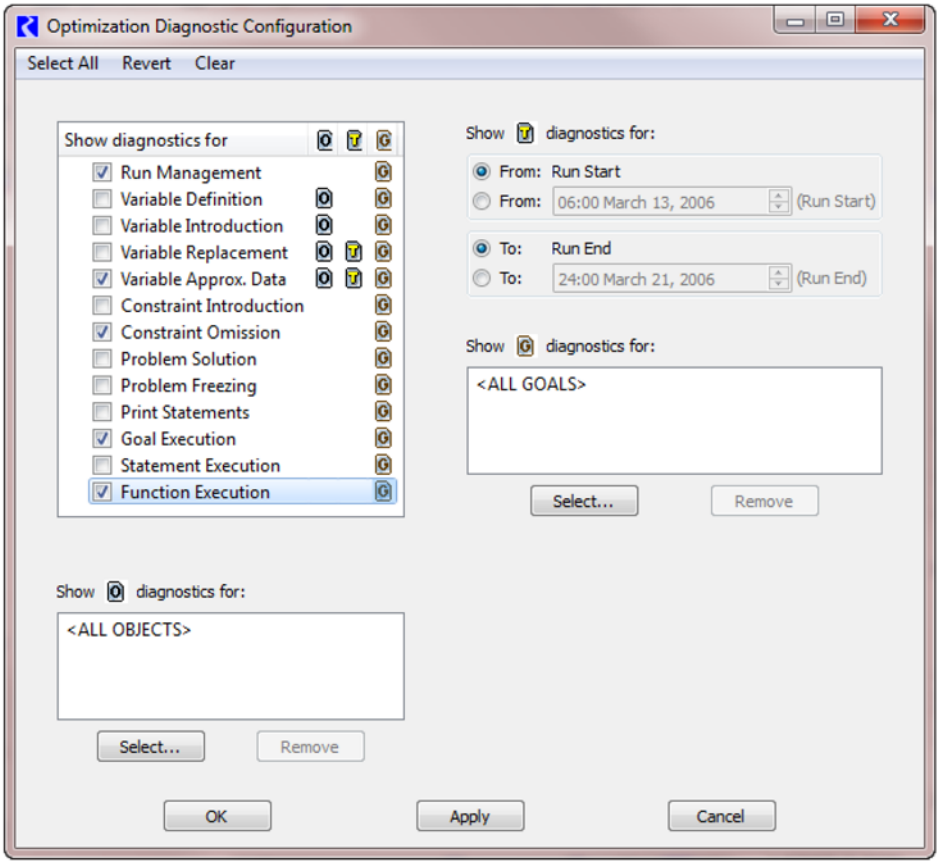
To enable or disable function diagnostics for many functions at once, from the RPL set editor select **Ruleset, then Function Diagnostics**. This will display a sub-menu which allows the following options:

- Enable Before Execution diagnostics for all user defined functions
- Enable After Execution diagnostics for all user defined functions
- Enable Before Execution diagnostics for all predefined functions
- Enable After Execution diagnostics for all predefined functions
- Disable Before Execution diagnostics for all user defined functions
- Disable After Execution diagnostics for all user defined functions
- Disable Before Execution diagnostics for all predefined functions
- Disable After Execution diagnostics for all predefined functions

Optimization Diagnostics

The Optimization Diagnostics Configuration dialog allows you to select from thirteen diagnostics groups and three context filters which are relevant during a Optimization run. This dialog's settings are only valid when the Optimization controller is selected. It is accessed from the Optimization button on the Diagnostics Manager.

Figure 3.11



Optimization Diagnostic Groups

Table 3.4 summarizes the types of messages that are displayed when a Diagnostic Group is selected.

Table 3.4

Diagnostics Group	Type of diagnostics issued
Run Management	Beginning/End of Run and each goal. **Examples:** • Initializing model run. • Beginning of run. • Begin goal. • Execute goal. • End goal • End of run.
Variable Definition	Diagnostic messages for definition of each variable: • Defining the variables of slot "SM3 reservoir.Pool Elevation" in terms of variables on the shifted slot "Shifted Pool Elevation" (shift = -320.00000000 "m^1"). • Adding definition "Shift Scaling of Pool Elevation" for the following variables: SM3 reservoir.Pool Elevation [t] • ... Definition: (SM3 reservoir.Pool Elevation [t] == (SM3 reservoir.Shifted Pool Elevation [t] - -320.00000000 "m^1"))
Variable Introduction	Diagnostic messages for introduction of new decision variables: • While preparing for the run, encountered the new decision variable "SM3 reservoir.Outflow [24:00 October 31, 2117]"; to define this variable 1 constraints will be introduced into the optimization problem. • While executing statement 3.1.1(date=24:00 October 31, 2092).1, encountered the new decision variable "SM3 reservoir.Opt piece for Power.4 [24:00 October 31, 2092]"; to define this variable 1 constraints will be introduced into the optimization problem.
Variable Replacement	Diagnostic messages for replacement of variables: • While executing statement 1.1.1(date=24:00 March 31, 2100).1, replacing one constraint with another. • ... Constraint being replaced: ((1.00000000 * SM3 reservoir.Shifted Pool Elevation [24:00 March 31, 2100]) <= 87.00000000 "m^1") • ... Replacement constraint: (SM3 reservoir.Storage [24:00 March 31, 2100] <= 12642800000.00000000 "m^3")
Variable Approx. Data	Diagnostic messages for approximation of data: • The constraint "((1.00000000 * Wilbur.Bypass Capacity [18:00 March 13, 2006]) >= 1.13267386 "m^3 / s^1")" was determined to be ineffective while attempting to substitute an equivalent constraint on "Wilbur.Storage" using the table "Wilbur.Bypass Capacity Table". The constraint has been ignored because it is redundant for all values in the table, which may need additional values. The nearest value in the table is 1.982179 cms. If this constraint was indirectly generated by RiverWare, prior diagnostic messages or turning on informational diagnostics may be helpful. This warning is issued for this constraint once per run, although this situation might hold for multiple dates. The data table is automatically generated at the beginning of each run based on "Wilbur.Bypass Table", so data corrections should be made there.

Table 3.4 (Continued)

Diagnostics Group	Type of diagnostics issued
Constraint Introduction	Diagnostic messages for introduction and linearization of constraints: • Adding the defining constraint "(Def.) Economic Value.Linear Avoided Operating Cost [24:00 September 30, 2085], Linear Avoided Operating Cost Replacement" to the optimization problem. • ... Original constraint: (SM3 reservoir.Power [24:00 June 30, 2110] >= 100.00000000 "MW") • ... Linearized constraint: (((((((3.00965758 "1000000.000000 kg^1 / m^1-s^2" * SM3 reservoir.Opt piece for Power [24:00 June 30, 2110]) + (2.98868019 "1000000.000000 kg^1 / m^1-s^2" * SM3 reservoir.Opt piece for Power.1 [24:00 June 30, 2110])) + (2.97411902 "1000000.000000 kg^1 / m^1-s^2" * SM3 reservoir.Opt piece for Power.2 [24:00 June 30, 2110])) + (2.95792886 "1000000.000000 kg^1 / m^1-s^2" * SM3 reservoir.Opt piece for Power.3 [24:00 June 30, 2110])) + (2.90415559 "1000000.000000 kg^1 / m^1-s^2" * SM3 reservoir.Opt piece for Power.4 [24:00 June 30, 2110])) + (2.77190824 "1000000.000000 kg^1 / m^1-s^2" * SM3 reservoir.Opt piece for Power.5 [24:00 June 30, 2110])) + (2.20884309 "1000000.000000 kg^1 / m^1-s^2" * SM3 reservoir.Opt piece for Power.6 [24:00 June 30, 2110])) >= 100.00000000 "1000000.000000 m^2-kg^1 / s^3")
Constraint Omission	Diagnostic messages when constrains are skipped as they will have no effect. • Attempt to add a soft constraint "C22.1.1(res=Wilson).1(date=12:00 March 21, 2006).1 70" which would not constrain the problem (e.g., it might be weaker than the restrictions imposed by the bounds on the variables), so it is not being added. The full constraint is: ((1.00000000 * Wilson.Outflow [12:00 March 21, 2006]) >= 0.14158423 "1000.000000 m^3 / s^1") • Skipping the new constraint C25.1.1(res=Apalachia).1(date=24:00 March 21, 2006).1 36 because the left-hand side is already bounded by other constraints (C11.1.1(res=Apalachia).1(date=24:00 March 21, 2006).1 1 A and C11.1.1(res=Apalachia).1(date=24:00 March 21, 2006).1 1 B). Currently, 67204983.66069999 "m^3" <= lefthand side <= 67204983.66069999 "m^3".
Problem Solution	Diagnostic messages for solving the optimization problem and the number of variables and constraints that are freezing • Freezing 0 variables. • Freezing 0 constraints. • While executing statement 2.1, defining a REPEATED MAXIMIN objective involving 2 constraints and solving the problem. • ... Objective: (100.00000000 * ZM2.1) • ... Original objective: (... • ... Linearized objective (...
Problem Freezing	Diagnostic messages for freezing variables and constraints • Frozen constraint: Ending Pool Elevation, C2.1.1(date=24:00 September 30, 2119).1 1 B: was 267.970000 <= ... <= 267.970000 (dual cost = 30.567953). Current statement: CONSTRAINT $ "SM3 reservoir.Pool Elevation" [date] == $ "Opt Data.Ending Elevation" [date] • Frozen variable: ZM3.1 = 0.100000 (reduced cost = 100.000000)

Table 3.4 (Continued)

Diagnostics Group	Type of diagnostics issued
Print Statements	User defined messages within rules for each goal. **Example:** • Minimum Load goal was successful.
Goal Execution	Diagnostic messages displaying when a goal is executed: • Executing goal #3 ("MinimumLoad", within group "Firm Energy")
Statement Execution	Diagnostic messages displaying when each statement within a goal is executed • Executing CONSTRAINT statement 3.1.1(date=24:00 September 30, 2119).1, which will attempt to add the soft constraint: (SM3 reservoir.Power [24:00 September 30, 2119] >= 100.00000000 "MW"). • Executing REPEATED MAXIMIN statement 1.1.
Function Execution	Diagnostics messages for function execution (both user defined and predefined). “Before Execution” function diagnostics show the function being called and the arguments passed into the function. “After Execution” function diagnostics show the value that the function evaluated to and whether or not the value was constrained. **Note:** Function diagnostics also need to be enabled on the functions themselves. This can be done through the Function Editor dialog (for enabling diagnostics on individual functions) or by selecting **Ruleset**, then **Ruleset Function Diagnostics** from the main **RPL Set Editor** (for enabling or disabling function diagnostics for all functions in the ruleset). See “Function Diagnostics” on page 57 for details. • GetMaxReleaseGivenInflow (% “WattsBar”, 0.634 [“1000 * cms”], @”Start Timestep”) • “GetMaxReleaseGivenInflow” evaluated to: 1299.413 [“cms”]
Init. Rules Rule Execution	Initialization Rules rule execution diagnostics are printed for each rule. For an Optimization run, there is no filtering based on rule name. When enabled, diagnostics are shown for all Initialization rules. **Examples:** • Executing rule (#3) Assignment initiated (the left-hand side is “Mohave.Storage”[]) Evaluation of Assignment statement successful; will attempt assignment Mohave.Storage[July, 1998] = 1590800.00 [1.00 acre-ft] Rule successfully finished • Early termination: NaN encountered. The problem was encountered at the following location within the expression: “FC Violation”(). • RHS of assignment evaluated to NaN.

Table 3.4 (Continued)

Diagnostics Group	Type of diagnostics issued
Init. Rules Function Execution	Diagnostics messages for function execution (both user defined and predefined) from initialization rules. For an Optimization run, there is no filtering based on rule name. When enabled, diagnostics are shown for all Initialization rules. "Before Execution" function diagnostics show the function being called and the arguments passed into the function. "After Execution" function diagnostics show the value that the function evaluated to and whether or not the value was constrained. **Note:** Function diagnostics also need to be enabled on the functions themselves. This can be done through the Function Editor dialog (for enabling diagnostics on individual functions) or by selecting **Ruleset**, then **Ruleset Function Diagnostics** from the main **Ruleset Editor** (for enabling or disabling function diagnostics for all functions in the ruleset). See "Function Diagnostics" on page 57 for details. **Examples:** • GetMaxReleaseGivenInflow (% "WattsBar", 0.634 ["1000 * cms"], @"Current Timestep") • "GetMaxReleaseGivenInflow" evaluated to: 1299.413 ["cms"] • "GetMaxReleaseGivenInflow" constrained to: 1200.00 ["cms"]
Init. Rules Print Statements	User defined messages for each initialization rule. For an Optimization run, there is no filtering based on rule name. When enabled, diagnostics are shown for all Initialization rules. **Example:** • Entering rule curve routine for January - July

Workspace Diagnostics

Note: This document is under development.

Table 3.5 summarizes the types of messages that are displayed when a Diagnostic Group is selected.

Table 3.5

Super Group	Diagnostics Group	Type of diagnostics issued
Model Load/Save:		
Data Management		
	DMI Management	
	DMI Invoke	
	DMI Dataset	
	DMI Slot	
	Rule DMI	
Workspace Management:		
Controller Management:		Any changes to controller settings: • Controller set to "Rulebased Simulation".

Table 3.5 (Continued)

Super Group	Diagnostics Group	Type of diagnostics issued
Rule Management Groups:		
Subbasin Management		Any changes in Subbasin management • Add subBasin ("New Subbasin 1").
Object Management		Any changes to an Object on the workspace • OBJECT: MuddyLake: Rename object to "MuddyReservoir".
Slot Management		
Link Management		
Dependency Management		
Name Map Management		
Client Server		
MRM Rule Execution		For the iterative MRM ruleset, this prints information on MRM rule, timestep, and slot. This is similar information to Rule Execution; see “Rule Execution” on page 53.
MRM RPL Function Execution		For the iterative MRM ruleset, prints before and after execution diagnostics for user and predefined functions. This provides similar information to Function Execution diagnostics; see“Function Execution” on page 53.
MRM RPL Print Statements		For the iterative MRM ruleset, this allows print statements to post diagnostics. This is similar information to Print Statement; see “Print Statements” on page 54.
MRM Rule DMI		For the iterative MRM ruleset, this posts information on DMI’s that are executed by MRM Rules.

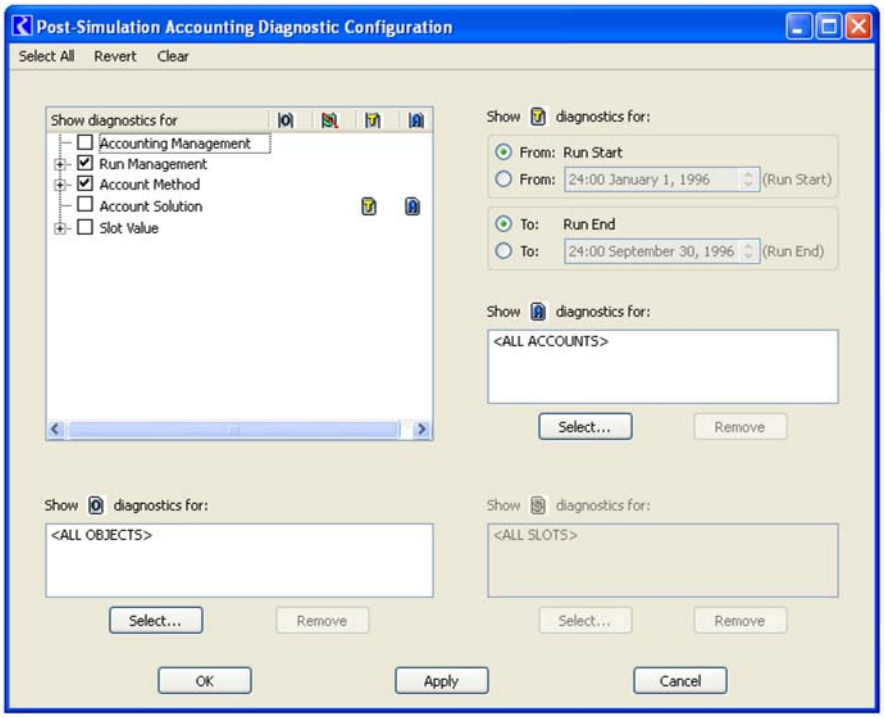
Table 3.5 (Continued)

Super Group	Diagnostics Group	Type of diagnostics issued
	Expr. Slot Execution	Expression slot execution diagnostics are printed for expression slots that are evaluated outside of a run by selecting **Evaluation**, then **Evaluate** on the slot or **Control**, then **Evaluate Expression Slots** from the workspace. **Note:** There is no timestep filtering on the workspace diagnostics, so these diagnostics will print a message for all timesteps in which the expression slot evaluates. **Example:** • Expression evaluated to 624.00 [1000 acre-ft]
	Expr. Slot Function Execution	Evaluation of RPL functions called by expression slots evaluated outside of a run. Diagnostics messages are printed for function execution (both user defined and predefined). “Before Execution” function diagnostics show the function being called and the arguments passed into the function. “After Execution” function diagnostics show the value that the function evaluated to and whether or not the value was constrained. **Note:** Function diagnostics also need to be enabled on the functions themselves. This can be done through the Function Editor dialog (for enabling diagnostics on individual functions) or by selecting **Set**, then **Function Diagnostics** from the main **Set Editor** (for enabling or disabling function diagnostics for all functions in the set). The Expression Slot Set editor is accessed from the Workspace **Policy**, then **Open Expression Slot RPL Set**. See “Function Diagnostics” on page 57 for details on configuring the diagnostics. See “Series Slots With Expression” in *User Interface* for details on expression slots. **Example:** • SumFlowsToVolume($”Mountain Storage.Inflow”, @”Start Timestep”, @”t”) • “SumFlowsToVolume” evaluated to: 274839.74 [“m3”].

Accounting Diagnostics

Figure 3.12 shows the Post-Simulation Accounting Diagnostics Configuration dialog.

Figure 3.12



Available filters include Objects, Accounts, Slots, and Timesteps. Diagnostics allow the user to see when each individual account is solving, what slots are known and unknown at that timestep, the accounting method being used, what the value is when an account's slot is set, and more.

Accounting Diagnostic Groups

Table 3.6 summarizes the types of messages that are displayed when a Diagnostic Group is selected.

Table 3.6

Super Group	Diagnostics Group	Type of diagnostics issued
Run Management:		
	Controller	Beginning/End of Run/Timestep for each time. **Examples:** • Initializing model run. • Beginning of Run. • Begin Timestep. • Execute timestep. • End of run.

Table 3.6 (Continued)

Super Group	Diagnostics Group	Type of diagnostics issued
	SimObj	Beginning/End of Run/Timestep for each Object and time. **Examples:** • Clear state. • Propagating user input. • Beginning of run. • Begin timestep. • End timestep. • End of run.
	Account	Beginning of Run for each Account and time. **Examples:** • Clear state. • Propagating user input.
Account Method:		
	Controller	Describes when the controller is notified of a change to an account's slot. **Example:** • The "Post-Simulation Accounting" controller received a slot changed notification for slot "BigLake^City.Inflow".
	SimObj	Describes which user defined methods are being executed. **Example:** • Executing Category "Storage Account Gain Loss" with selected method "BigLake_Gain Loss".
	User Method Execution	Diagnostics for object level accounting methods. **Examples:** • Executing user-defined accounting method #3 ("HappyLake_SlotInflows", within group "Storage Account Slot Inflow") • Assignment initiated (the left-hand side is "$ "HappyLake^Fish.Slot Inflow" []"). • Evaluation of Assign statement successful; will attempt assignment: Fish.Slot Inflow[January 1, 2004] = 3.99 [1.00 * cfs].
Independent Groups:		
	Account Management	• N/A
	Account Solution	Describes when the account is solving including the knowns and unknowns. **Example:** • Inflow is known, Storage is allowed, Outflow is input; solving for Storage.
Slot Value:		

Table 3.6 (Continued)

Super Group	Diagnostics Group	Type of diagnostics issued
	Set Value	Describes when an accounting slot is set. **Example:** • Confluence^Fish.Inflow: Set value = 25.740254 * 1.0cfs
	Interpolation	TableSlot table interpolation lookups for each account slot.

Chapter 4
RPL Debugging and Analysis

Topics

- Types of RPL Debugging and Analysis, page 69
- RPL Debugger, page 83
- RPL Diagnostics, page 98
- Rulebased Simulation Model Run Analysis Tool, page 101
- RPL Analysis Tool, page 101

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.

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.

Errors When Building RPL Expressions

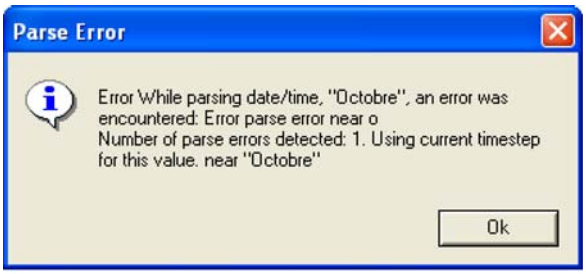
When an unspecified expression is filled in by entering text into its text field, the entry is parsed to ensure that it meets the requirements of the desired type. If the entry is not valid, a diagnostic message is generated. For example, entering `@"t"` into an unspecified <numeric expr> would generate the following message shown in Figure 4.1 and revert the expression to its prior, unspecified, state.

Figure 4.1



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 message shown in Figure 4.2 and revert the expression to its prior, unspecified, state.

Figure 4.2



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

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 accuracy 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 as follows.

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

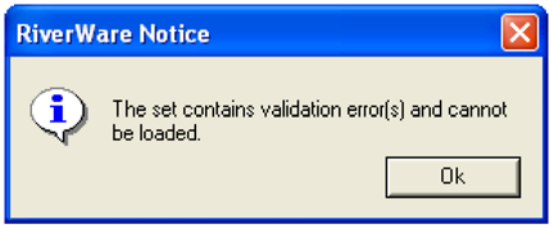
An *evaluatable* RPL set is one which, as far as can be determined without actually evaluating it, could be successfully evaluated. This means that all of the previous validity levels have been satisfied. This also means that there cannot be any remaining unspecified expressions in the RPL set and that 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, the window shown in Figure 4.3 appears, indicating the location within the file where the error was detected.

Figure 4.3

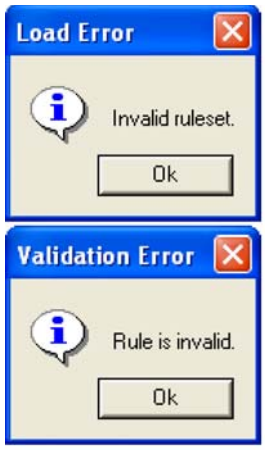


A message will also appear in the Diagnostics Output Window indicating that the "RPL set loading failed" and provide the path and name of the RPL set. The Diagnostics Output Window will display the line number in which the parsing error occurred. If this should occur, first verify that the file you are 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 selecting 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 shown in Figure 4.4.

Figure 4.4



Another message appears in the Diagnostics Output Window describing the validation errors. 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.

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:

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 dependent slots change.

Example 4.1 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.

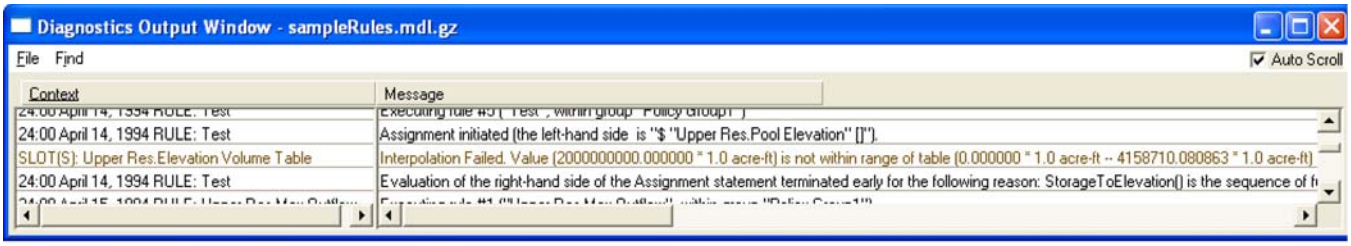
Pool Elevation	Storage
895.00	0
920.00 . . . 1060.00	811,000 . . . 8,241,000
1085.00	10,233,000
1110.00	12,452,000
1135.00	14,920,000

The predefined function, StorageToElevation(), will do the following:

- 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.

  Note: There is nothing wrong with the StorageToElevation() function itself. The arguments that 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 dependent 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 4.2 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.

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 dependent 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 4.3 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 ( “Main 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 4.4 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 = FLOW x 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 4.5 Example:

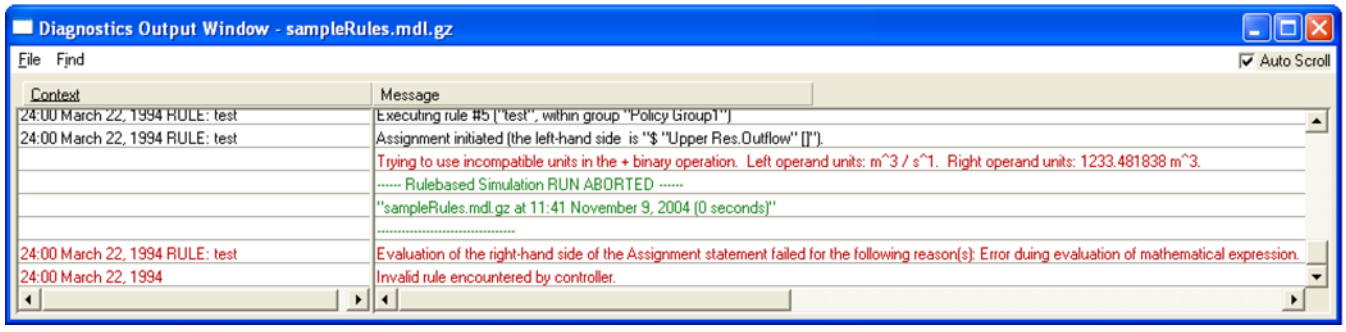
The mathematical expressions below may not be evaluated because the unit types of their elements are inconsistent.

```
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, such as the one shown in Figure 4.5.

RPL unit type errors are only detected during evaluation.

Figure 4.5



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 4.6 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 reasons: 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 addition 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 4.7 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”.

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 and 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 4.8 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.MinRes-
   Release[]
   ```

2. Set Rio Outflow for Maximum Fish Flow

   ```
   Rio.Outflow[] = IF ( Rio.Outflow[] > DataObj.MaxFishFlow[] ) THEN DataObj.MaxFish-
   Flow[]
   ```

3. Set Rio Outflow to Meet Downstream Demands

   ```
   Rio.Outflow[] = DataObj.TotalDownstreamDemand[]
   ```

The Rulebased Simulation proceeds as follows:

1. Initially, no values are known, so Rio cannot dispatch.
2. Rule #3 fires and sets Rio.Outflow to meet demands.
3. Rio dispatches with the Solve given Outflow, No Routing dispatch method based on these priorities:
   - Inflow = unknown
   - Outflow = known at priority 3R

   and solves for Inflow at controller priority 3
4. 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 3R Outflow with the new priority 2R Outflow.

5. Rio redispatches with the Solve given Outflow, No Routing dispatch method based on these priorities:
   – Inflow = known at priority 3
   – Outflow = known at priority 2R

   and solves for Inflow at controller priority 2.
6. 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 2 Inflow with the new priority 1R Inflow.
7. Rio redispatches with the Solve given Inflow, No Routing dispatch method based on these priorities:
   – Inflow = known at priority 1R
   – Outflow = known at priority 2R

   and solves for Outflow at controller priority 1. This causes Rule #2 to go back on the agenda because of a new value for Outflow.
8. Rule #2 fires and sets Rio.Outflow for maximum fish flow. The rule can overwrite the existing priority 1 ("O" flag) Outflow with the new priority 2R Outflow.
9. Rio redispatches with the Solve given Inflow, No Routing dispatch method based on these priorities:
   – Inflow = known at priority 1R
   – Outflow = known at priority 2R

   and solves for Outflow at controller priority 2, the last rule that set a slot. 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 generated

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 ineffectively. 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 custom slot 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 4.9 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 Outflow, then the Diversion Request is set on the UpperRioDiversion. As in the previous example, UpperRio Inflow may be overwritten by a minimum release rule.

The ruleset used to control the Rios is as follows.

1. Set UpperRio Inflow due to Minimum Upstream Reservoir Release
   ```
   UpperRio.Inflow[] = IF ( UpperRio.Inflow[] < DataObj.MinResRelease[] ) THEN Data-
   Obj.MinResRelease[]
   ```
2. Set UpperRioDiversion Not to Exceed UpperRio.Outflow
   ```
   UpperRioDiversion.Diversion Request[] = MIN (DataObj.UpperRioDivRequest[], Upper-
   Rio.Outflow[])
   ```
3. Set LowerRio Outflow to Meet Downstream Demands
   ```
   LowerRio.Outflow[] = DataObj.TotalDownstreamDemand[]
   ```

The Rulebased Simulation proceeds as follows:

1. Rule #3 fires and sets LowerRio.Outflow to meet demands.
2. LowerRio dispatches with the Solve given Outflow, No Routing 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.
3. UpperRio dispatches with the Solve given Outflow, No Routing dispatch method based on these priorities:
   – Inflow = unknown
   – Outflow = known at priority 3

   but cannot solve completely because the UpperRioDiversion is not yet known.
4. Rule #2 fires and sets the UpperRioDiversion Diversion Request equal to the UpperRio Outflow (assuming that the request in the Data Obj is very large).
5. UpperRioDiversion dispatches and sets the Diversion from UpperRio at controller priority 2. This propagates across the link to UpperRio.Diversion.
6. UpperRio redispatches with the Solve given Outflow, No Routing 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.
7. 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.

8. UpperRio redispatches with the Solve given Inflow, No Routing 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.
9. LowerRio redispatches with the Solve given Inflow, No Routing 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.

10. Rule #2 fires and resets the UpperRioDiversion 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.
11. UpperRio redispatches with the Solve given Inflow, No Routing dispatch method based on these priorities:
   - Inflow = known at priority 1R
   - Outflow = known at priority 1
   - Diversion = known at priority 2

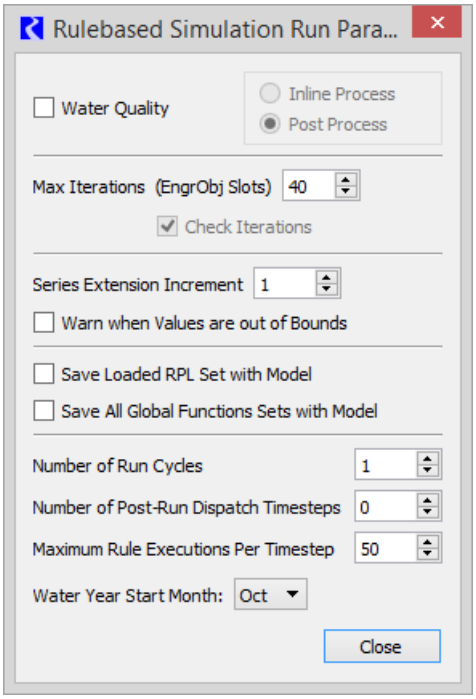
   and solves for a new UpperRio.Outflow at controller priority 2. The dispatch can overwrite the existing priority 1 Outflow with a new priority 2 Outflow. This value propagates across the link to LowerRio's Inflow. This also causes Rule #2 to be put back on the Agenda.
12. LowerRio redispatches with the Solve given Outflow, No Routing 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 UpperRio.Outflow.
13. UpperRio redispatches again with the Solve given Inflow, No Routing 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 Lower-Rio's Inflow.
14. 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 dimmed box for Check Iterations in the Rulebased Simulation Run Parameters dialog. From the **View** menu of the Run Control dialog, select **Rulebased Simulation Run Parameters...** to open the dialog. Within this dialog, the user can change the number of maximum iterations (the default number of maximum iterations is 40). For more complex models, the user may be required to increase the number of maximum iterations.



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 the Maximum Rule Executions Per Timestep specified in the Rulebased Simulation Run Parameters dialog. The default maximum executions is 50. From the **View** menu of the Run Control dialog, select **Rulebased Simulation Run Parameters...** to open the dialog to change this value. 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 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 the maximum executions, 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 4.10 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:

```
1. Data.A[] = IF ( Data.B[] == 1 ) THEN
       0
   ELSE
       1
```

2.
```
Data.B[] = IF ( Data.A[] == 1 ) THEN
       1
    ELSE
       0
```
3.
```
Data.A[] = 1
```

The Rulebased Simulation would proceed as follows:

1. Rule #3 fires and sets Data.A to 1.
2. Rule #2 fires and sets Data.B to 1.
3. Rule #1 fires and sets Data.A to 0. Rule #2 goes back on the Agenda.
4. Rule #2 fires and sets Data.B to 0. Rule #1 goes back on the Agenda.
5. Rule #1 fires and sets Data.A to 1. Rule #2 goes back on the Agenda.
6. The steps above are repeated another 24 times before the Rulebased Simulation Controller stops the run.

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. See “RPL Debugger” on page 83 for details.
- **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 Statements. See “Rulebased Simulation Diagnostics” on page 50 for details.
- **Rulebased Simulation Model Run Analysis tool:** Provides information on the rules and policy that caused an object to dispatch. See “Rulebased Simulation” on page 25 for details.
- **RPL Set Analysis Tool:** Provides information about the number of executions and execution times for RPL blocks and statements. See “RPL Analysis Tool” on page 101 for details.

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 on diagnostics and the model run analysis tool.

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.

About 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 RPL logic 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:

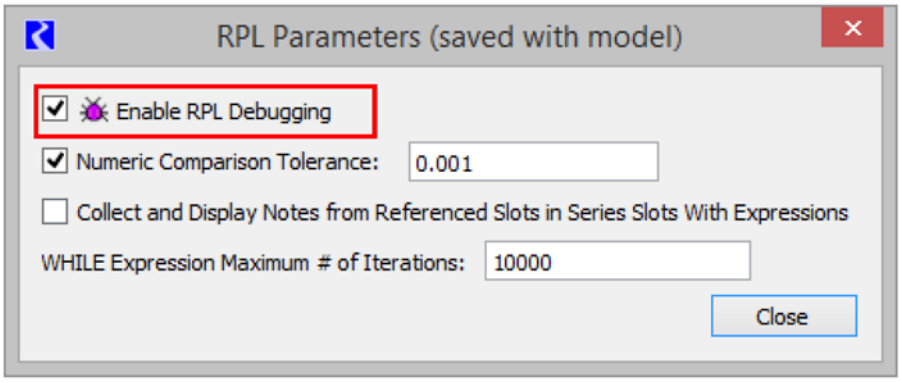
- Before a statement in a rule, optimization goal, or user-defined accounting method is executed
- After all statements in a rule, optimization goal, or user-defined accounting method have executed
- Before a function’s body expression is evaluated
- After a function’s body expression is evaluated
-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

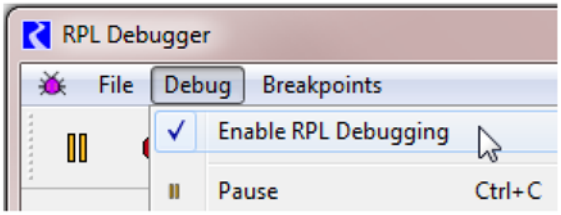
Enabling RPL Debugging

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

- In the RPL Parameter dialog (**Policy**, then **RPL Parameters**), select the **Enable RPL debugging** checkbox.

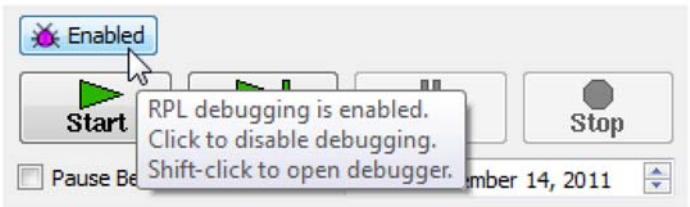


- From the RPL Debugger (**Policy**, then **RPL Debugger),** use the **Debug**, then **Enable RPL Debugging** menu checkbox.



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. Select the button to select or clear 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**, then **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.

See "RPL Debugger Scenario 1" on page 96 for a scenario in which the user might want to disable and then enable debugging temporarily.

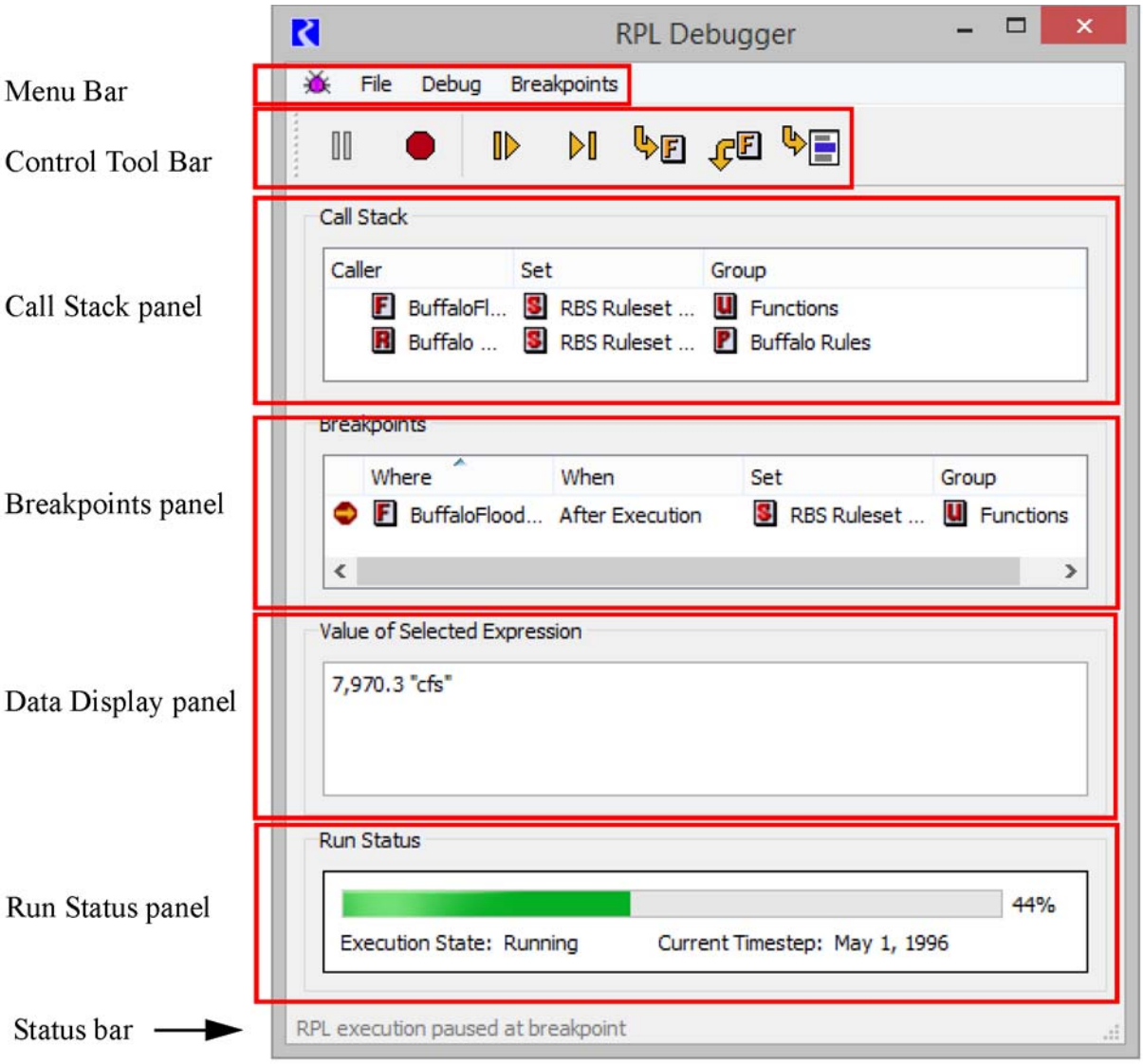
RPL Debugger Dialog

The RPL Debugger dialog is accessible as follows:

- From the Workspace **Policy** menu (**Policy**, then **RPL Debugger**)
- From any RPL dialog **Block** (**Set, Rule, Goal, Method**, **Function**, etc.), then **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. Figure 4.6 shows the principle components of this dialog.

Figure 4.6



- Menu bar. Menu access to debugger functionality.
- Control toolbar. 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; see "Error Handling" on page 95 for details.
- Run Status panel. Shows the overall run status including the current controller timestep.
- Status bar. Brief description of the state of the debugger.

Menu Bar

Table 4.1 describes the functionality available through the **File** and **Breakpoints** menus. See “Control Toolbar and Debug Menu” on page 88 for details about **Debug** menu.

Table 4.1

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

Control Toolbar 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. Table 4.2 describes the button controls in the control toolbar and **Debug** menu of the Debugger dialog.

Table 4.2

Menu Item	Button	Keyboard accelerator	Description
Enable RPL Debugging			Turn on or off the collection of RPL See “Enabling RPL Debugging” on page 85 for debugging information.
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.

Table 4.2

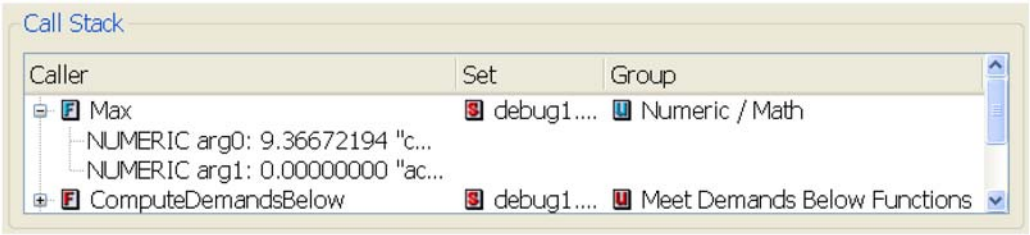
Menu Item	Button	Keyboard accelerator	Description
Continue		F5	When execution is paused, continue execution until one of the following occurs: • A breakpoint is reached. • The user requests another pause or stop event. • RPL execution terminates normally.
Step		F10	When execution is paused, continue 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.
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**, then **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.

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. Figure 4.7 illustrates.

Figure 4.7

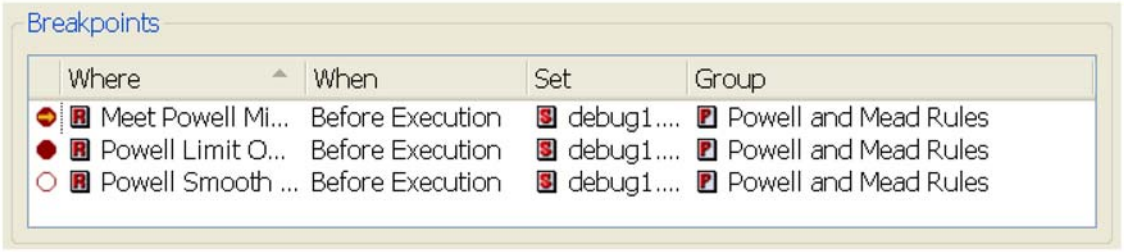


Breakpoints Panel

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

- Icon. Red octagonal breakpoint indicator, filled indicates that it is enabled , unfilled indicates disabled . Selecting the stop sign indicator will enable/disable the breakpoint.
- Where. 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. Group (where applicable) that contains that item.
- Set. 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.

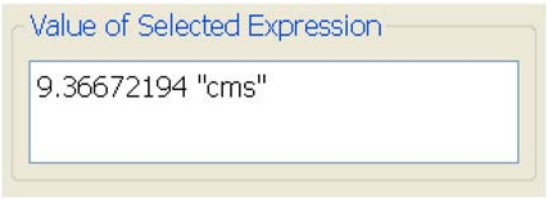
When RPL execution pauses at a breakpoint, the associated row in the Breakpoints Panel is scrolled into view.

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; see “Sample Use Scenarios” on page 96 for details. Within the RPL Debugger dialog, they can be deleted and/or temporarily disabled/enabled using the Breakpoints menu. Also, selecting the stop sign indicator will enable/disable the breakpoint. Breakpoints are persistent (saved with the RPL item which contains the breakpoint).

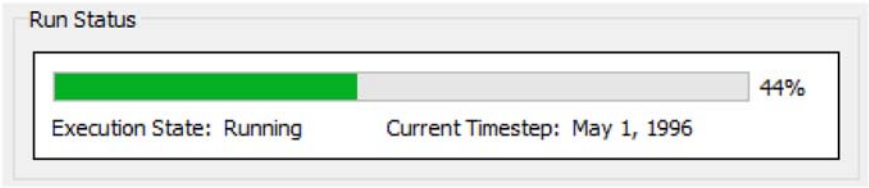
Value of Selected Expression Panel

When paused or after a run, the Value of Selected Expression panel displays the value of the selected RPL expression. See "Displaying Data Values" on page 93 for details.



Run Status Panel

The Run Status panel at the bottom of the debugger shows the overall status of the run. This information is also shown on the Run Control / Run Status dialog. The panel in the debugger provides easy access to the current controller timestep.



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. See "Sample Use Scenarios" on page 96 for a description of this scenario.

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.

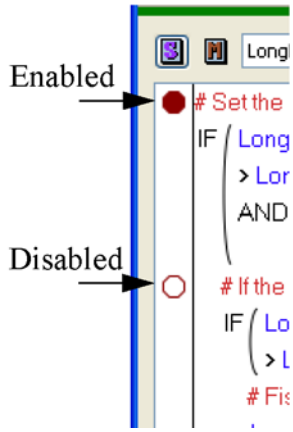
Selecting the margin adds a breakpoint (break *before*) at that location if there is not one already; disables it if there is an enabled one, and deletes it if there is a disabled breakpoint. That is, selecting the margin cycles through three states, enabled, disabled, and deleted. If the select location 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)**, then **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.

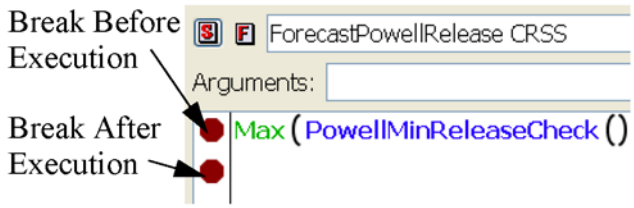
If there is a breakpoint associated with a given location, a solid red octagon is drawn in the debugging margin (unfilled red octagon for disabled breakpoints). For blocks, the breakpoint indicator is associated with a specific statement or appears just below the last statement (indicating a breakpoint which is activated after the last statement has executed). Figure 4.8 illustrates.

Figure 4.8



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. Figure 4.9 illustrates.

Figure 4.9



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:** 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, as shown in Figure 4.10.

Figure 4.10



Debug cursor in a function at Break Before Execution



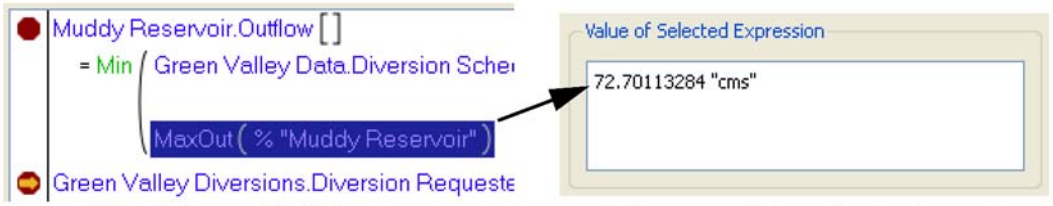
Debug cursor after function execution (no breakpoint)

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

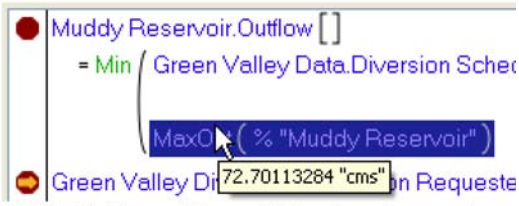
expressions are not named, the user can not pick an expression for display by entering its name or by selecting it in a list of expression names; rather the most straightforward method is to select the expression in the RPL editor in which it is displayed. Once selected, there are two ways to look at 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

Data Value Units

The value displayed uses the settings in the Unit Schemes; see "Unit Schemes" in *User Interface* for details. 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 365 days in the year.

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; see "Control Toolbar and Debug Menu" on page 88 for details.

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 select **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 select **Continue** on the RPL debugger. The timestep then finishes and the Run Control will pause before the next timestep.

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

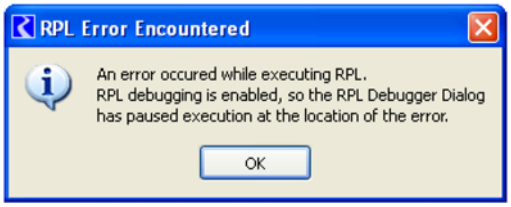
2. The debugger will not display the value of the entire left hand side of an assignment statement. Technically, the left hand side is not evaluated, it receives the value to which the right hand side evaluates, so it does not have a value to display. If you wish to see the value that will be assigned, 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.

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:

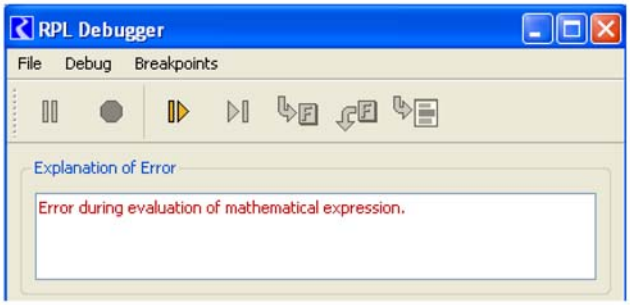
- A statement in a block
- The body of a function
- The expression of an expression slot.

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. 

Sample Use Scenarios

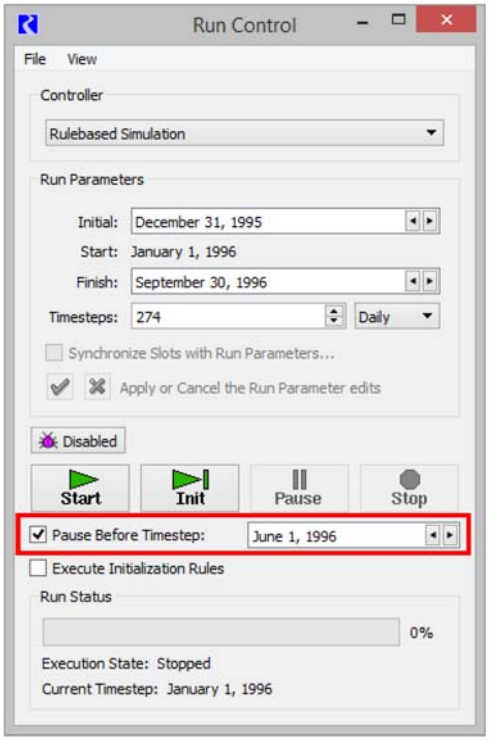
In this section we present a few example scenarios that illustrate different ways to make use of the RPL debugger.

RPL Debugger 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:

1. Disable RPL debugging; see “Enabling RPL Debugging” on page 85.
2. Configure the Run Control to pause before the desired date, as shown in Figure 4.11.

Figure 4.11



3. Start a run and let it run until it pauses at the timestep indicated.
4. Enable RPL debugging.
5. 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.
6. Continue or step the run from the Run Control.

   The run proceeds until a breakpoint is reached.
7. Examine RPL values as desired and use the debugger's **Step** and **Continue** buttons to control RPL execution.
8. When finished, delete or disable breakpoints and continue in the RPL debugger and in the Run Control to finish. Or, select the **Stop** button in the debugger to stop RPL execution and end the run.

RPL Debugger 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, use the following steps.

1. Enable debugging but do not set any breakpoints.
2. 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; see "Error Handling" on page 95 for details.

3. Determine what the error is by using debugging tools.
4. Select **Continue** in the debugger and the run will stop.
5. Fix the error in the RPL dialog.

Note: The error message is repeated in the diagnostics output.

RPL 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; see "About RiverWare Diagnostics" on page 33.
- Rulebased Simulation; see "Rulebased Simulation Diagnostics" on page 50.
- RPL Expression Slots and MRM RPL sets; see "Workspace Diagnostics" on page 63.

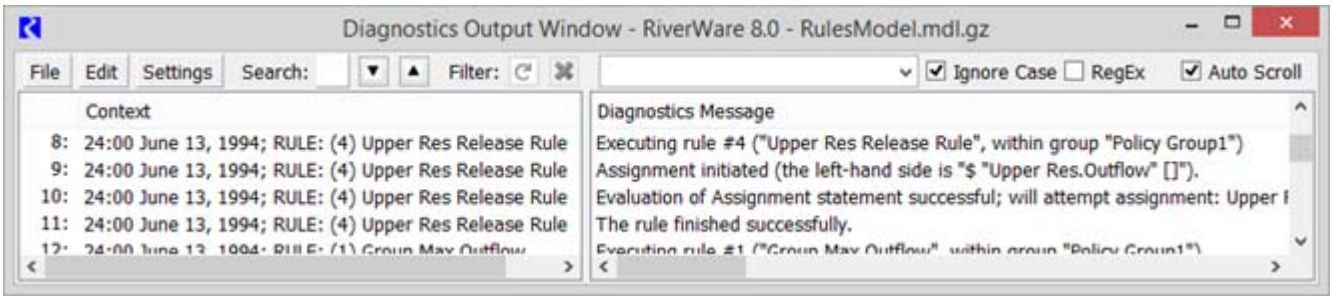
Units for RPL Diagnostics

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. See "Unit Schemes" in *User Interface* for details. For monthly and annual values, however, 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.

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.

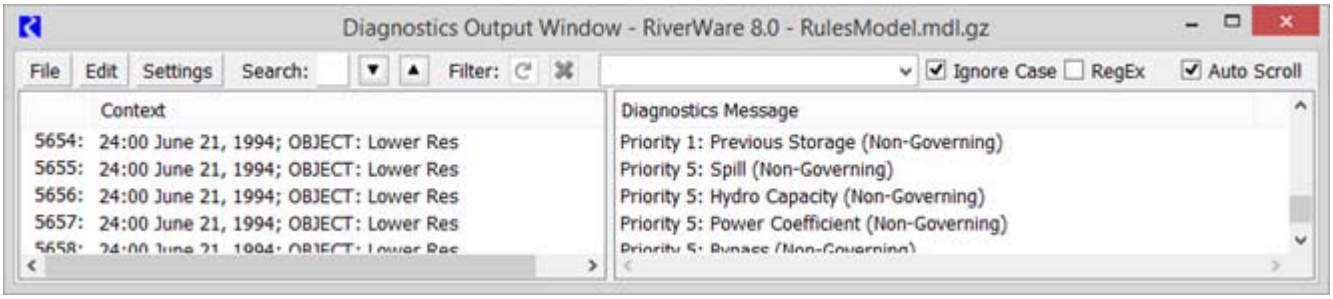
- The Rule Execution diagnostics group provides information about the evaluation of rules, including when they fire, whether they evaluate successfully, whether they attempt to set a slot, and the value which they attempt to set.



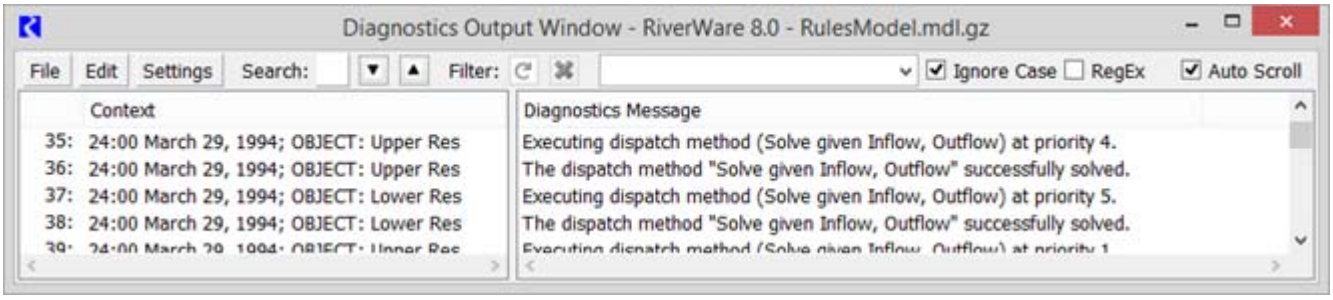
- The **Print Statements** diagnostics group enables the printing of PRINT statements in rules. If this diagnostics group is not turned on, no PRINT statements will be displayed in the Diagnostics Output Window.



- The **Dispatch Management**, then **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.

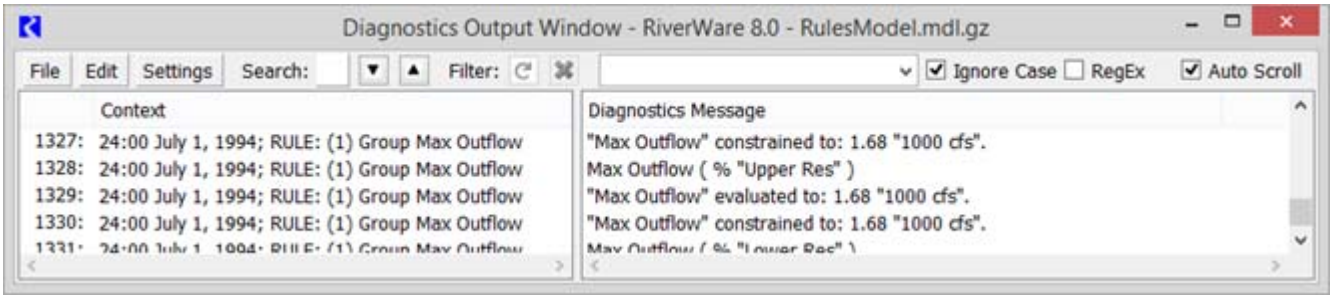


- The **Dispatch Management**, then **Controller** diagnostics group generates a message whenever an object begins dispatching and provides the controller priority during the dispatch.



- 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 diagnostics show when the function is

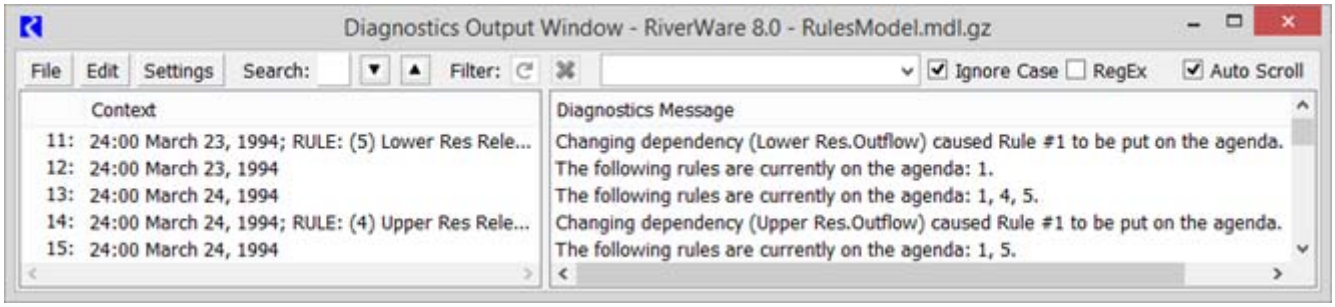
called and provides the arguments passed into the function. After Execution diagnostics show the result from the function call and any post-execution constraints imposed.



- The **Rule Management**, then **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.

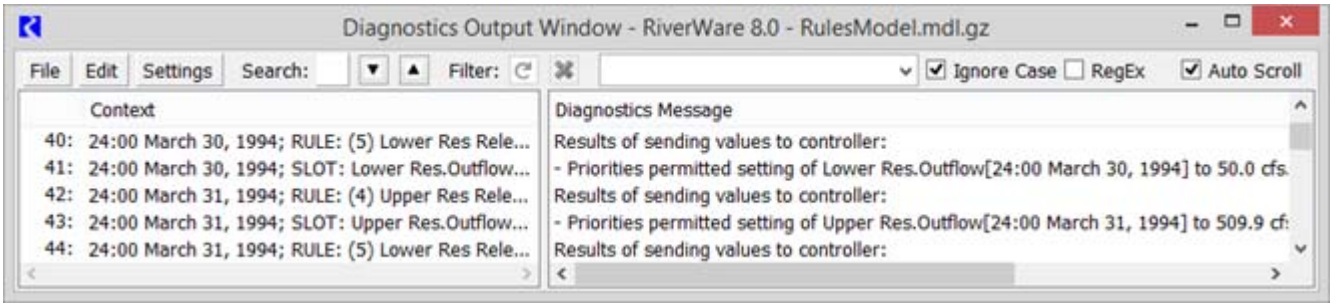


- The **Rule Management**, then **Agenda** diagnostics group provides information about the state of the Agenda. After each rule executes, this diagnostics group posts a list of all rules on the Agenda. The diagnostics group also posts a message whenever a rule is placed back on the Agenda due to a change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



- The **Rule Management**, then **Assignments** 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.



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.

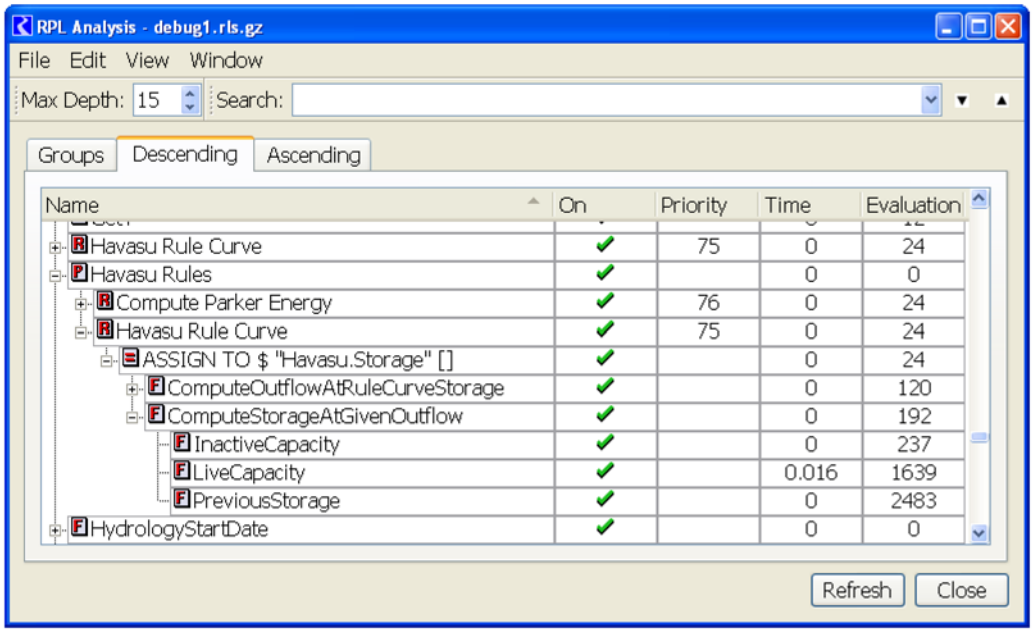
See "Rulebased Simulation" on page 25 for a description of the rulebased simulation Model Run Analysis tool.

RPL Analysis Tool

The RPL analysis tool assists the user in analyzing and documenting a RPL set. For this section, *items* refers 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.

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.

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**, then **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**, then **Objects** menu) is: sets, groups, blocks (rules), print statements, statements, user functions, and predefined functions.

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.

Using the RPL Analysis Dialog

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

Opening the 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**, then **Analysis**

- From the RPL group editor: **Group**, then **Analysis**
- From the Rule editor: **Rule**, then **Analysis**
- From the Function editor: **Function**, then **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.

Switching Between Views

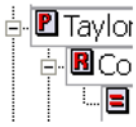
As the three listviews provide different views on the same set of RPL items, it is often helpful to switch between these different views during a session. As long as the **Window**, then **Sync Views** is active by default), the three listviews will focus on the same selected item in all three views.

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**, then **Groups View** or **Ctrl+G**, **View**, then **Descending View** or **Ctrl+D**, **View**, then **Ascending View** or **Ctrl+A**.

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.



- 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**, then **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**, then **Collapse All** or **Alt + Left-arrow** will collapse all the rows in the current listview.
- **View**, then **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**, then **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.

Sorting

As with many other listviews in RiverWare, each listview can be sorted by a specific column. Selecting 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.

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 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 selecting 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.

Pressing a letter on the keyboard will navigate to the next item that begins with the given letter; see "Navigating Within a Treeview" on page 104 for details.

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 an item in a listview.
- **Edit**, then **Open Editor** or **Alt+E**.

For cases when the user is opening a large number of editor dialogs, the **Window**, then **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.

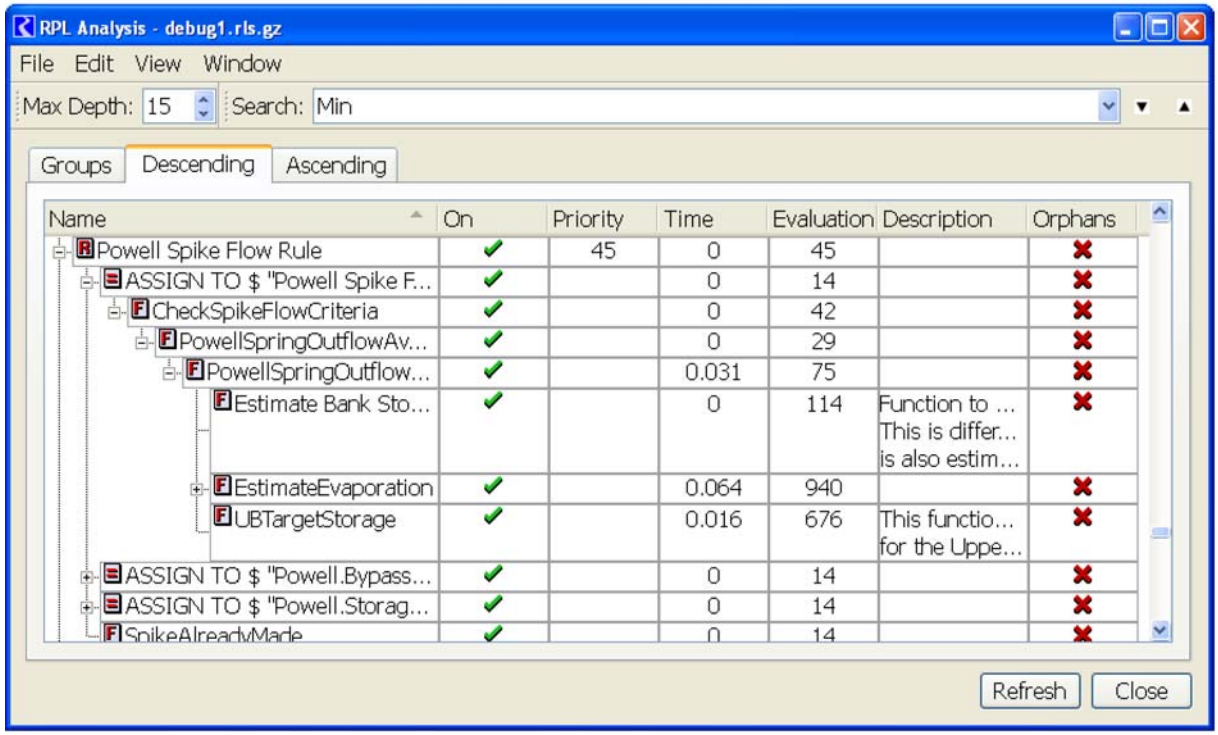
Customizing the Views

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.

As shown in Figure 4.12, the dialog can display a large amount of information. For most tasks, however, 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.

Figure 4.12



Selecting the Columns to Display

The **Window**, then **Columns** menu 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 the item in the treeview, using the key command **Alt+E**, or selecting **Open Editor** from the context-menu or **Edit**, then **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**, then **Objects** menu 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.

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

File, then **Print**, then **Selected** and **File**, then **Export**, then **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.

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**, then **Sync Views**. When this is active, selecting an item in one of the listviews will also select it in the other two listviews.
- **Window**, then **Replace Existing Editor**. When this is active, opening a RPL editor (by double-clicking an item or using the **Edit**, then **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**, then **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]**, then **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 the menu bar and selecting **Depth Toolbar**.

