Debugging

DevOps Model RealTime Models

Mattias Mohlin
Senior Software Architect
HCL

This document describes how to debug a UML-RT model created in DevOps Model Real-Time.

The document was last updated for Model RealTime 11.3. All screen shots were captured on the Windows platform.

Introduction

Debugging a model that has been transformed into a real-time application helps you to

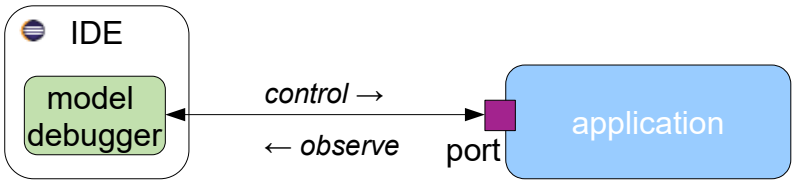
- track down and fix run-time problems and logical errors in the application
- understand the run-time behavior of the application
- visualize the current state of the application
- test how the application handles unusual scenarios

Debugging a model in Model RealTime is similar to debugging a C++ program using CDT or a command-line debugger such as gdb. In fact, you could accomplish all the above tasks by only using a C++ debugger. However, that approach would have some drawbacks:

- it requires you to map constructs in generated C++ code back to the model
- it requires debugging into the TargetRTS code, and a deep understanding of how it implements the RT Services Library
- it may require writing special debug code to, for example, send events to a capsule instance, or to capture events received at a port
- remote debugging of an application that is deployed on a target machine can be complex (need to have access to e.g. gdbserver on the target machine)

The model debugger in Model RealTime is not a replacement for a C++ debugger, but should rather be seen as a complement to it. It allows you to debug the application at model-level rather than code-level. If you wish you can combine both model-level and code-level debugging at the same time. This can be very powerful since it allows you to execute the application primarily at model-level, but then use the C++ debugger to do more low-level tasks such as inspecting the values of variables, step into external C++ code, look at current threads etc.

The model debugger works by starting a generated application in **observability** mode. The application then opens a socket on a port which the Model RealTime IDE can connect to. Using this communication channel Model RealTime can query the application for run-time information, such as the currently available capsule instances and their current states. It can also send debug commands to the application, for example to suspend or resume its execution, add or remove breakpoints, send events to ports etc. Hence the model debugger allows you both to **control** the execution of the application, as well as to **observe** what happens when it executes. The picture below illustrates how the model debugger in the Model RealTime IDE interacts with the debugged application.



Starting a Model Debug Session

You can start a model debug session in three different ways:

- Automatically launch the application locally and immediately start to debug it

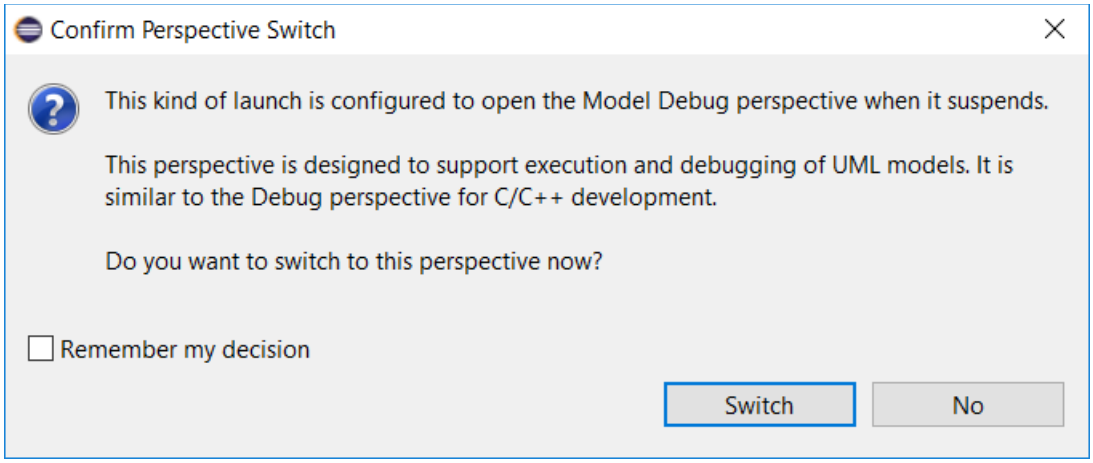
- Automatically launch the application locally, and at a later point in time start a debug session by attaching to the running application
- Manually launch the application locally or remotely (e.g. when it has been deployed on a target machine) and start a debug session by attaching to the running application

Let's look at each of these alternatives in more detail below.

Debug As RealTime Application

To automatically launch the application on your local machine, and immediately start to debug it, follow these steps:

1. Right-click on the transformation configuration (TC) which you used for building the application. Perform the command **Debug As – RealTime Application**. Note that the TC must be of type Executable.
2. This command will first build the TC (if necessary) and then launch the built application on your local machine. The application will start in a suspended mode where it waits for commands from the model debugger. When the application has been launched, and the model debugger is successfully connected to it, you will be prompted to switch to the Model Debug perspective.



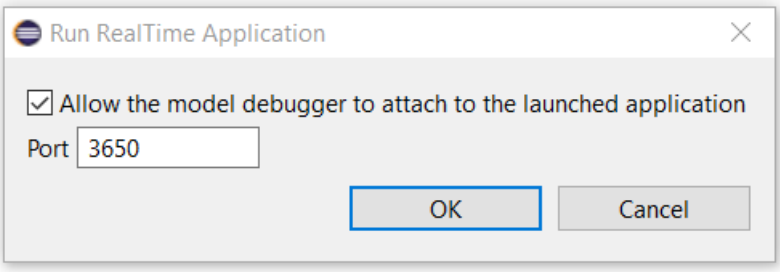
It's recommended to use the Model Debug perspective while debugging the application at model-level.

You are now ready to start debugging the application using the commands provided by the Debug view.

Run As RealTime Application

Sometimes you may not want to debug the application from the beginning, but instead let it run for a while and then at some later point in time start to debug it. The main reason would be performance; an application runs somewhat slower when the model debugger is attached to it. So if your application needs to run for a while before it enters a state in which you want to debug it, then you may want to use this approach:

1. Right-click on the TC which you used for building the application and perform the command **Run As – RealTime Application**. The TC must be of type Executable.
2. In the dialog that appears, mark the checkbox **Allow the model debugger to attach to the launched application**.
   Also change the port number, if necessary.



When you press **OK** the application will start to run in observability mode. This means that you can then later attach the model debugger to the running application by following the steps described in Attach the Model Debugger to a Running Application. The application will start to run immediately, i.e. it will not wait for you to attach the model debugger right away.

Note that if you don't mark the checkbox in the above dialog, the application will run in normal mode, without any possibility to attach the model debugger to it. This can be useful if you just want to run the application and observe its behavior using other ways than the model debugger, for example by looking at printed trace messages.

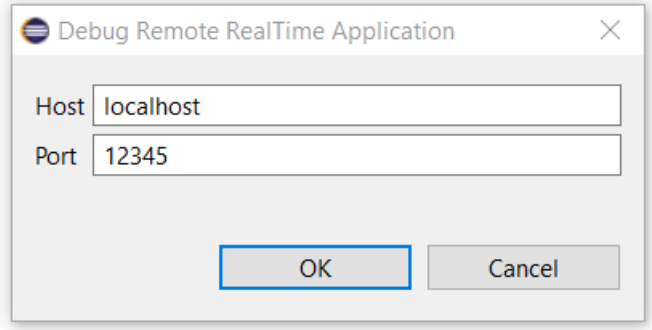
Attach the Model Debugger to a Running Application

The model debugger can attach to any application that runs in observability mode. Running the application as descibed in Run As RealTime Application is one way to start the application in observability mode. But you can also start the application in this mode manually from the command-line, by using the command line option `-obslisten=<port>`, where `<port>` is a port that is available on the machine. For example:

```
./executable -obslisten=12345
```

The application will start in a suspended mode and will wait for the model debugger to attach to it on the specified port. Follow these steps to attach the model debugger:

1. Right-click on the TC which you used for building the application and perform the command **Debug As – Remote RealTime Application (Attach)**.
2. In the dialog that appears, specify the name or IP adress of the host machine where the application runs, and the port that was given as argument to the `obslisten` option.

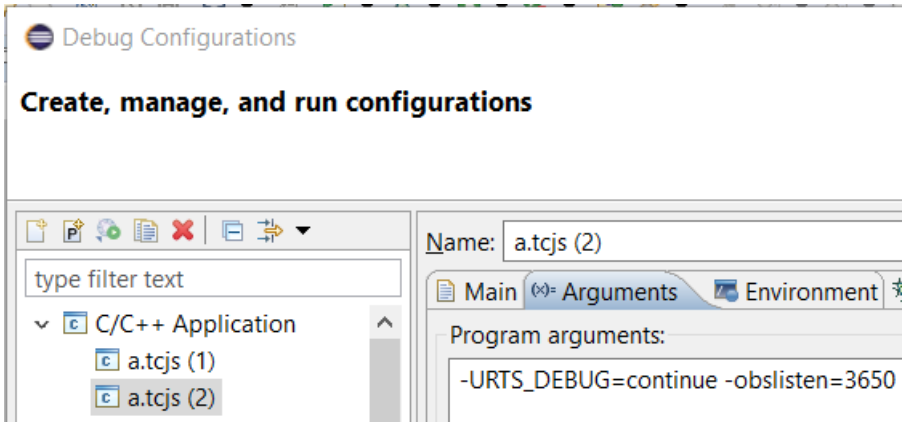


3. When the model debugger has successfully attached to the application, you will be prompted to switch to the Model Debug perspective.

You are now ready to start debugging the application using the commands provided by the Debug view.

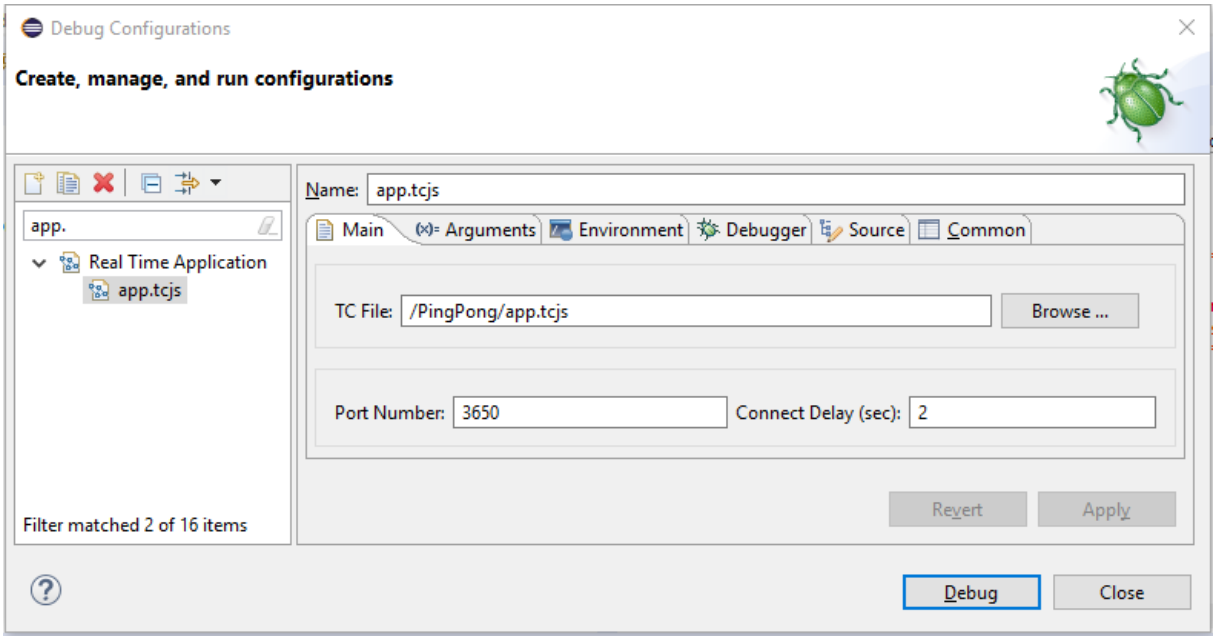
Launch Configurations

The commands in the ”Run As” and ”Debug As” context menus of a TC for starting a model debug session work by creating a so called **launch configuration** which contains all settings needed for launching the application, or to attach to an already running application. You can look at these launch configurations by means of the commands **Run – Run Configurations** or **Debug – Debug Configurations** respectively. The command that runs the generated application on the local machine generates launch configurations of type ”C/C++ Application”, with appropriate arguments specified in the Arguments tab. Which arguments that are used depend on if you want the model debugger to be able to later attach to the launched application or not. Here is what the launch configuration looks like when you launch the application with a possibility to later attach the model debugger to it:



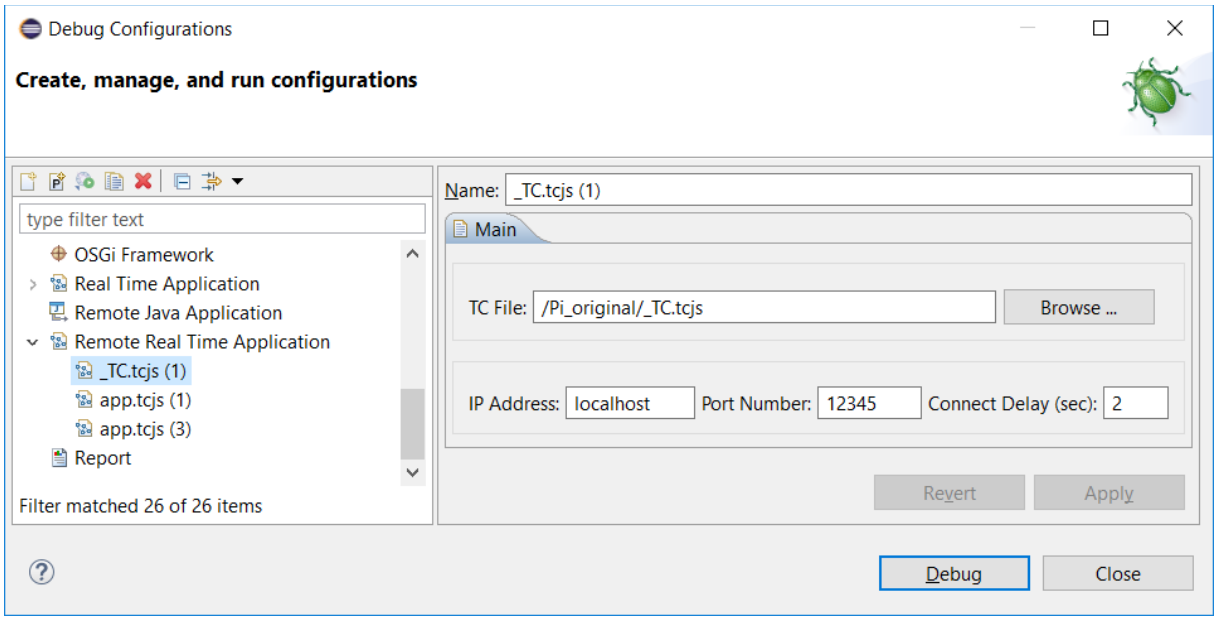
If you instead choose to run the application without a possibility to debug it, the argument will instead be `-URTS_DEBUG=quit` which effectively will disconnect the debugger from the application.

The command that launches the application locally and immediately starts to debug it generates a launch configuration of type ”RealTime Application”.



This type of launch configuration is actually very similar to the ”C/C++ Application” type. The only difference is that it contains a specification of the TC that built the application, and in general contains slightly fewer settings. Also, the port to use for communication between the model debugger and the application is specified on the Main tab and will automatically be translated to the argument `-obslisten=<port>`. The default port number is specified in the workspace preferences at *Run/Debug – RealTime Application*. You can override the default port number in a particular launch configuration, and this is for example necessary if you want to have multiple active debug sessions at the same time (either in the same or separate Model RealTime instances). In the same way you can override the default timeout (2 seconds) which specifies the maximum amount of time that the model debugger will wait for the launched application to start up. If your application starts slowly you may need to raise this limit.

The command that attaches the model debugger to an already running application generates a launch configuration of type ”Remote RealTime Application”.

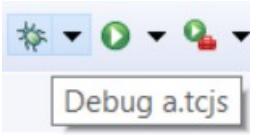


In this case the application is not launched by Model RealTime which means this type of launch configuration is much simpler than the others. All its settings are contained in the Main tab only.

In most cases you don't need to care about these generated launch configurations. They are created automatically, if necessary, when you use the context menu commands described above. However, as you can see, they contain some additional settings that you sometimes may need to adjust. For example, as already mentioned the default timeout when attempting to attach the model debugger to a running application is 2 seconds. If you have a slow connection to the target machine where the application runs, you may need to raise this limit.

Other situations when you need to make changes in the launch configurations include when your application takes custom command-line arguments, when environment variables need to be set for the launched process, when shared libraries are dynamically loaded from custom locations etc.

When you find a need to make changes in the generated launch configurations, you could consider to instead create your own launch configurations. These can be run or debugged directly from the launch configuration dialog (by pressing the **Run** or **Debug** button respectively). You can also use the toolbar buttons which are convenient if you want to re-launch a previously created launch configuration.



Press the arrow near these buttons to get a list of recently launched Debug or Run launch configurations. To repeat launching the most recently launched Debug or Run launch configuration just press the respective button once.

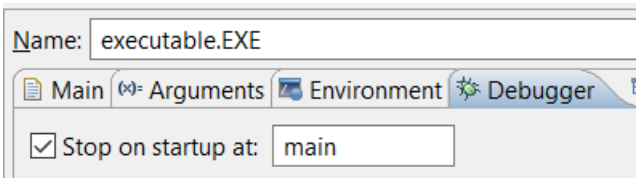
The table below summarizes the different launch configurations that are used and which command-line arguments that are set when launching the application.

Command	Launch Configuration	Command-line arguments
Debug As – RealTime Application	RealTime Application	`-obslisten=<port>`
Debug As – Remote RealTime Application (Attach)	Remote RealTime Application	`N/A (the application is not launched by Model RealTime)`
Run As – RealTime Application (not marking the checkbox ”Allow the model debugger to attach to the launched application”)	C/C++ Application	`-URTS_DEBUG=quit`
Run As – RealTime Application (marking the checkbox ”Allow the model debugger to attach to the launched application”)	C/C++ Application	`-URTS_DEBUG=continue, -obslisten=<port>`

Combining Model-Level and Code-Level Debugging

In Launch Configurations we saw that debugging an application locally works by starting it with a few special command-line arguments such as `-obslisten` and/or `-URTS_DEBUG`. It should therefore come as no surprise that it’s very easy to combine C++ level debugging with debugging at the model level. These are the steps needed:

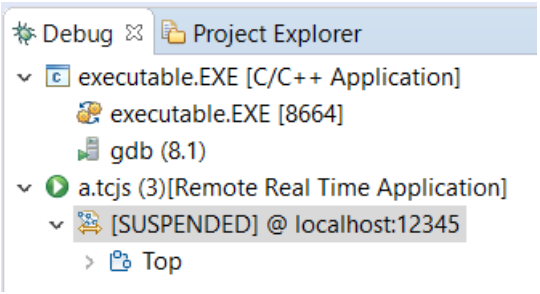
1. Create a Debug launch configuration of type ”C/C++ Application” for the generated C++ application.
2. Add the command-line argument `-obslisten=<port>` in the Arguments tab, where `<port>` is a port that is available on your machine.
3. Press **Debug** to start debugging the C++ application. By default you will be prompted to switch to the Debug perspective and the application will be suspended in the main function. If you don’t want it to stop in the main function (for example because you use the main function that is provided by the TargetRTS, and you don’t have a debug-compiled version of the TargetRTS), then you should unmark the checkbox in the Debugger tab.



4. Make sure that the application is not suspended. If it stopped in the main function you need to press the **Resume** button in the toolbar.
5. Now attach to the running application using the steps described in Attach the Model Debugger to a Running Application. Switch to the Model Debug perspective, or stay

in the Debug perspective. These perspectives are rather similar and either of them can be used in this scenario.

The Debug view now shows the application twice. It may look like this:



The first node represents the application at C++ level, while the second node represents the application at model level. Under the first node you will see information about threads, C++ call stacks and it integrates with the Variables view to let you inspect and/or edit variables when the C++ debugger has suspended the application. Under the second node you will see the structure of the real-time application at the model level of abstraction, and you can use it as explained in Debug View.

Running a combined C++ debugger and model debugger session can be very powerful since it allows you to execute the model primarily at model-level, but then use the C++ debugger to do more low-level tasks such as inspecting the values of variables, step into external C++ code, look at current threads etc. Some users may find it more convenient to use a separate Eclipse instance for the C++ debug session. The benefit with that approach is that you then have two separate Debug views (one in each Eclipse instance) so there is less need for scrolling up and down in this view to alternate between the C++ and model view of the debugged application. If you prefer to debug with something else than Eclipse CDT, for example Visual Studio, then you will always have a separate IDE instance for the C++ debug session, and only have the model debug session in the Model RealTime IDE.

When you debug the application using a C++ debugger you may want to step into TargetRTS code. To make this possible you need to use a version of the TargetRTS that contains debug symbols. Refer to the article in online help at *Model RealTime User's Guide – Articles – Running and Debugging – Debugging the RT services library* for details on how to build a debug version of the TargetRTS.

Debug View

The Debug view is the primary view used when debugging a real-time application with the model debugger. It allows you to control the execution of the application by commands such as Suspend and Resume. It also shows the current structure of the application in terms of which capsule instances that currently exist, which capsule parts they are located in, and which port instances they have. Other debug-related views and editors typically interact with the Debug view, either by looking at its currently selected node, or by supporting drag-and-drop of items onto the nodes.

Application Structure Visualization

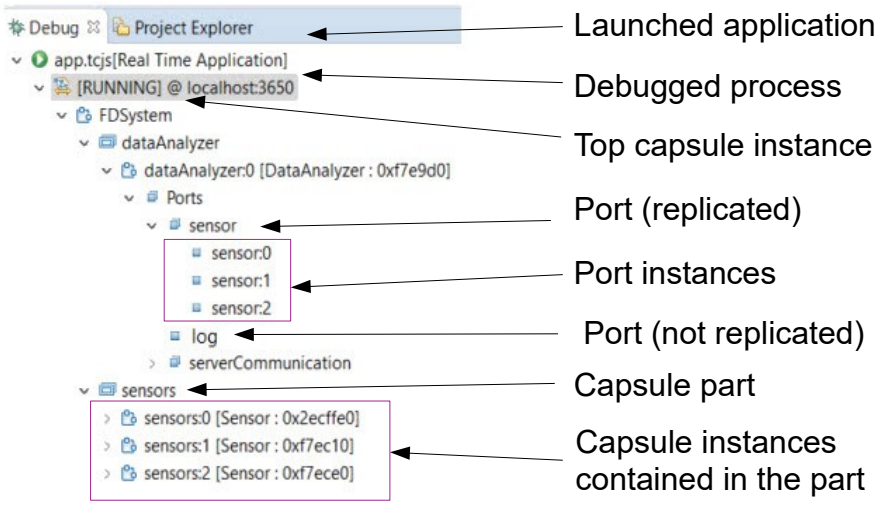
A debugged application is represented by a tree of nodes in the Debug view. The root node represents the application as a whole. Its label shows which launch configuration that was used for launching it (name and type). If the launch configuration was automatically generated by Model RealTime its name is the name of the TC that built the application (possibly followed by a number to make the name unique).

Below the root node there are nodes representing the processes in the application (only one in case the application is not distributed). The process node shows whether the application is suspended, running or terminated. It also shows the name of the host machine where the process runs, as well as the port it uses for communicating with the model debugger.

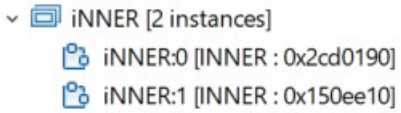
Below the process node there is a node that represents the instance of the top capsule, as specified in the TC.

If you expand further below the top capsule node you will see its capsule parts and below them the capsule instances they currently contain. Under a capsule instance node you can also see the ports of the capsule. If the port is replicated (i.e. has non-single multiplicity) all the connected port instances are shown below the port node.

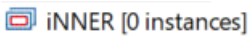
Below is a picture that summarizes the structure of the Debug view when debugging an application with the model debugger:



For a capsule part the current number of capsule instances it contains are shown in square brackets after the name. For example:



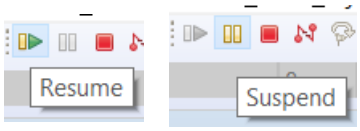
An empty capsule part is marked with a red rectangle in its icon:



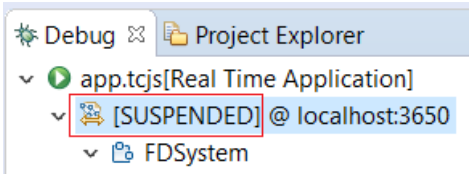
Note that the Debug view supports multiple simultaneous debug sessions, which means the Debug view can contain more than one root node. Most commands in the Debug view (and also some commands in other debug-related views) are sensitive to which node that is selected, so make sure the selection is correct before you perform a command. This is especially important in case you debug multiple applications at the same time.

Application Execution Control

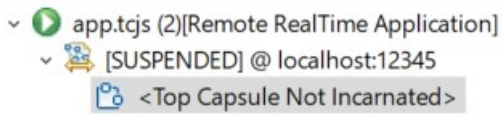
A debugged application can be in two states; **Running** or **Suspended**. When it is running you can perform the command Suspend to suspend it, and when it is suspended you can perform the command Resume to make it run again. The Suspend and Resume commands are available as buttons in the toolbar, and also as commands in the context menu of Debug view nodes.



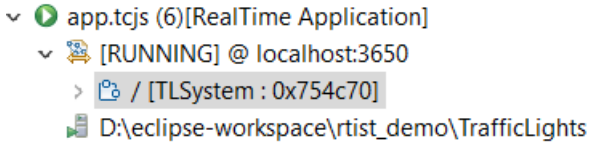
The current state of the application is shown on the Process node in the Debug view:



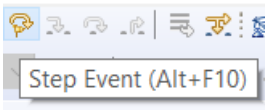
Note that when a debug session starts the application is initially suspended at a point where the top capsule is still not incarnated. However, the Debug view then already shows the top capsule. This is done so you can prepare the debug session before the application starts to run. For example, you can open the state instance diagram of the top capsule (see Instance Diagrams). The Debug view uses a special label for the top capsule node to show that it's not yet incarnated:



When you resume the application the label will change to show the information about the incarnated top capsule instance (capsule name and address of the capsule instance):

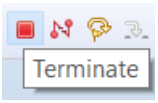


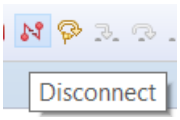
Another execution control command is **Step Event**.



It works by processing one event that is ready to be dispatched from the event queue of one of the controllers in the application. If there is no event to dispatch this command does nothing. For some kinds of applications this command allows you to single step through the application by processing one event at a time. However, in most cases it is easier to put a breakpoint at some interesting place in the application (see Breakpoints) and then resume the application to let it run until the breakpoint is hit.

When you want to stop debugging the application you can perform the command **Terminate**. Alternatively you can use **Disconnect**.



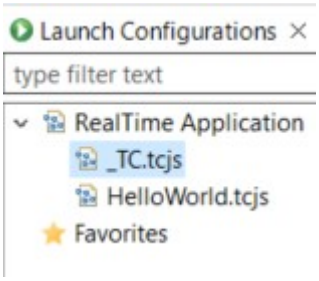


The difference between these commands is that Terminate will kill the debugged application, while Disconnect will just disconnect the debugger from it and let it continue to run. Disconnect is therefore mostly useful when you have attached the model debugger to an already running application that should continue to run after the debug session is finished.

In the context menu of Debug view nodes there are a few additional commands that sometimes are convenient:

- **Terminate and Relaunch**
  Terminates the application and then immediately relaunches it with the same settings as used previously.
- **Relaunch**
  Launches the application once more. This command is useful when you previously have terminated the application and then want to launch it again. However, it can also be used without first terminating the application and in that case it will lead to two instances of the application being debugged at the same time. This is normally not useful, but if you want to do it remember to first edit the launch configuration so that each debug session uses a unique port number.
- **Remove All Terminated**
  Removes all terminated debug sessions from the Debug view. Use this command to clean up the Debug view before you start another debug session.
- **Terminate and Remove**
  Terminates the application and then removes it from the Debug view.
- **Edit <TC>**
  Opens a dialog that lets you edit the launch configuration that was used for launching the debugged application. This can for example be convenient before you relaunch the application if you want to use slightly different settings (for example different command-line arguments or a different debug port).

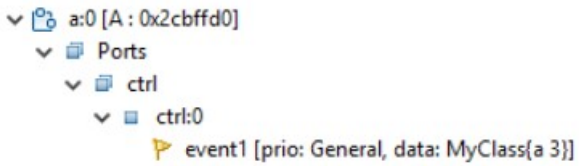
Many of these commands are also available in the Launch Configurations view:



From this view you can see all your launch configurations and launch them by double-click.

Event Queues

The Debug view can also show the event queues of a capsule instance, i.e. the events that have been sent to the capsule instance but that are not yet dispatched to it. Such an ”incoming” event is shown below the port instance node to which it was sent. Here is an example:

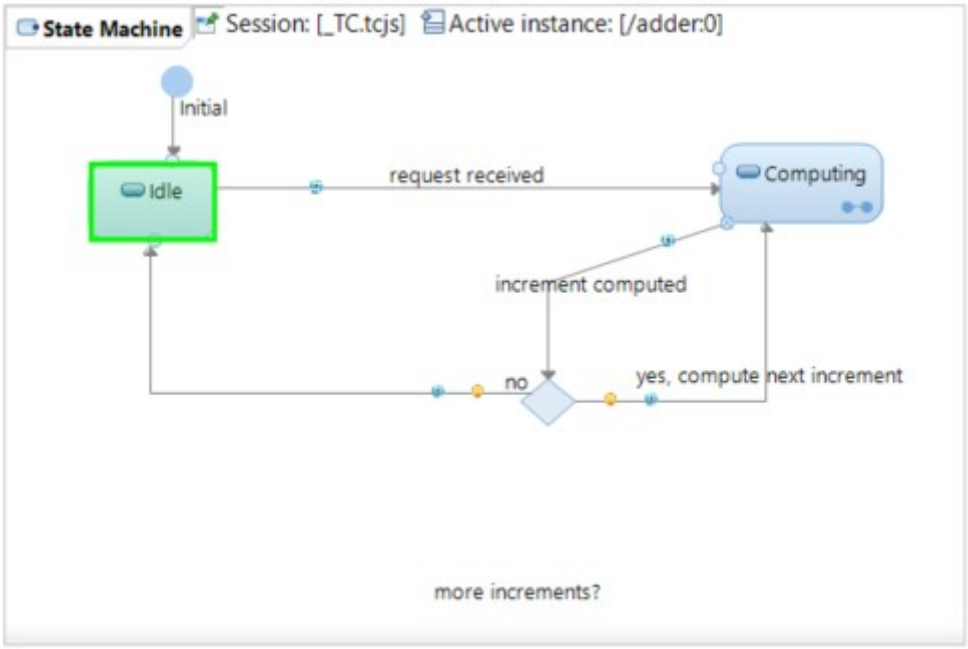


Here we can see that the event ”event1” has been sent to the capsule instance ”a:0” on its ”ctrl:0” port, and has not yet been handled. We can also see the priority and data (if any) of the event.

Contrary to other nodes in the Debug view, event nodes are only shown when the application is suspended. They will also appear immediately after you have manually injected an event using the Events view. However, when the application is running such an event is usually processed so quickly so you will not have time to see it unless the application is suspended when the event is sent.

Instance Diagrams

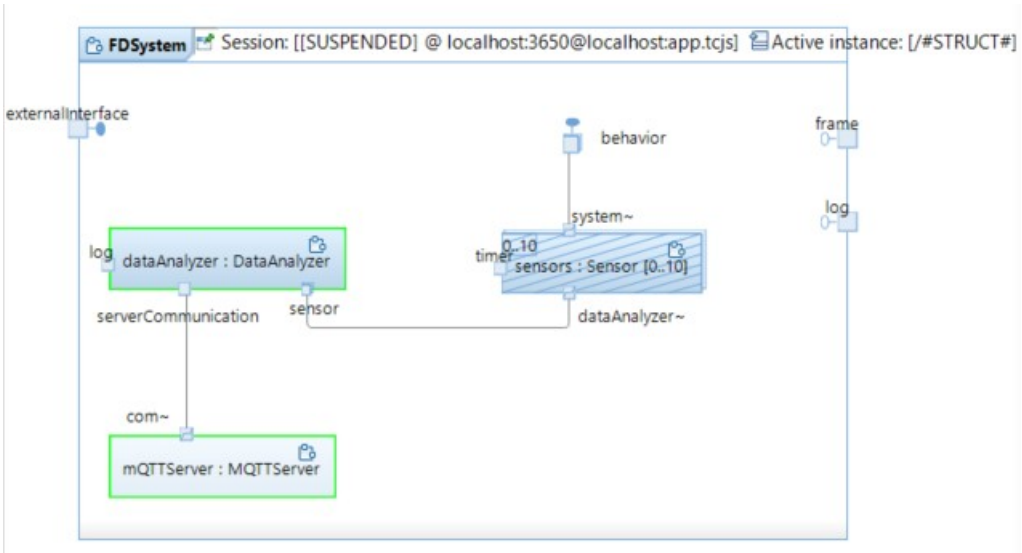
At run-time each instance of a capsule has its own state machine. While debugging it’s therefore necessary to use so called **instance diagrams** for inspecting a particular capsule instance. The current state (or current state configuration in case of a hierarchical state machine) can be seen in a state instance diagram. You can open it from the context menu of a capsule instance shown in the Debug view. The command is called **Open State Instance Diagram**. Here is an example of a state instance diagram:



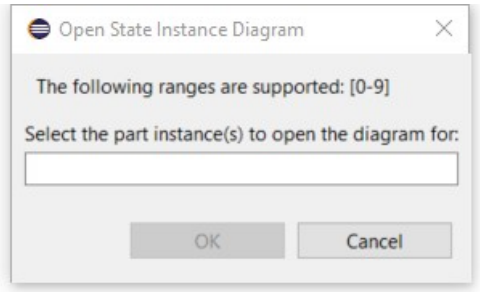
At the top of the diagram there is information about the capsule instance to which the diagram belongs. It also shows to which debug session it belongs, since it's possible to have multiple active debug sessions for the same application.

The active state is marked with a green flashing frame. In case of a hierarchical state machine, each state in the active state configuration is marked like that. Double-click on composite states in a state instance diagram to go into the state instance diagram that shows the nested state machine. You can configure the color of the "active state frame" using the preference *Run/Debug – RealTime Application – Active element color*. You have to restart the debug session for this preference to take effect.

There is a similar command **Open Structure Instance Diagram** for opening the composite structure diagram of a capsule instance. It shows which capsule parts that currently contain at least one capsule instance. Here is an example:

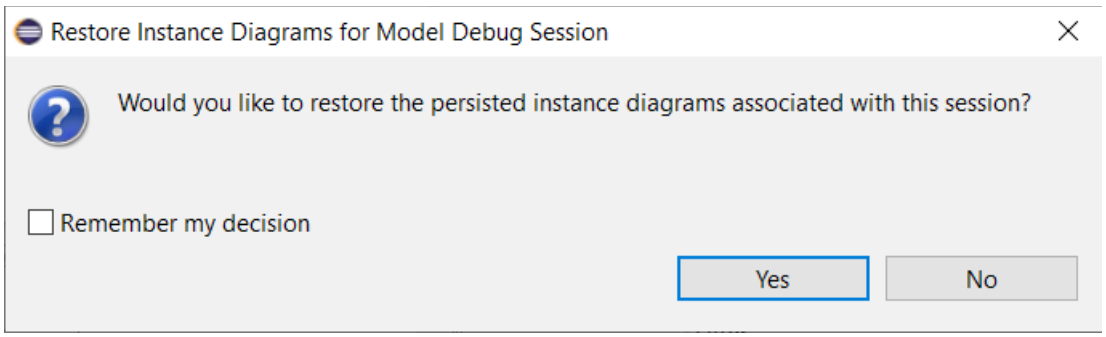


Just like for state instance diagrams the context menu commands on a capsule part that open other diagrams (**Open State Machine Diagram** and **Open Composite Structure Diagram** respectively) will open the corresponding instance diagram. If the capsule part is optional or has non-single multiplicity you will be prompted for which of the contained instances you want to open the diagram. For example, when the ”sensors” capsule part in the above example has been fully incarnated with 10 instances, the following dialog will appear if we try to open a composite structure or state machine diagram from it:



You can open multiple instance diagrams at once if you specify a comma-separated list of indices in this dialog (e.g. 0, 5, 14) or specify a range (e.g. 0-5).

The instance diagrams that you open during a debug session are remembered so that they can be automatically opened the next time you launch a debug session for the same application. You will be prompted when you start a model debug session if there are instance diagrams that could be automatically restored.

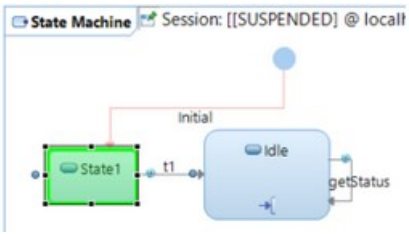


If you mark the checkbox your choice will be remembered the next time you start a model debug session. You can get back the dialog later by setting the preference *Run/Debug - RealTime Application - Restore instance diagrams for model debug sessions* to "Prompt".

Marking of Executed Elements

Sometimes it’s interesting to see which parts of the application that have executed so far. Model RealTime can visualize this either by marking executed elements with a special icon or a color. Use the preference *Run/Debug – RealTime Application – Mark executed elements* to enable the tracking of executed elements and how they should be visualized.

Executed elements are shown in the instance diagrams. Here is an example of a state instance diagram where we can see that only the initial transition has executed so far:

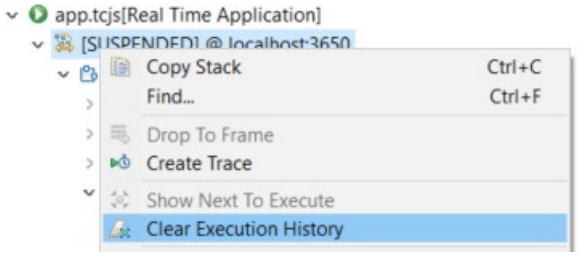


Visualizing executed elements can help to answer questions such as

- which parts of a state machine have executed at a certain point in time?
- how big is the coverage of an automatic test that has exercised the debugged application?
- is there any “dead” code in a state machine (unreachable states for example)?

Executed elements are only shown in state instance diagrams, not in structure instance diagrams.

The visualization of executed elements is removed when you terminate the debug session. If you want to remove it before that you can use the command **Clear Execution History** that is available in the context menu of the Process node in the Debug view:



This command will clear all collected information about executed elements. It can for example be useful if you want to visualize which parts of a state machine that were executed when sending a particular event to a port.

Breakpoints

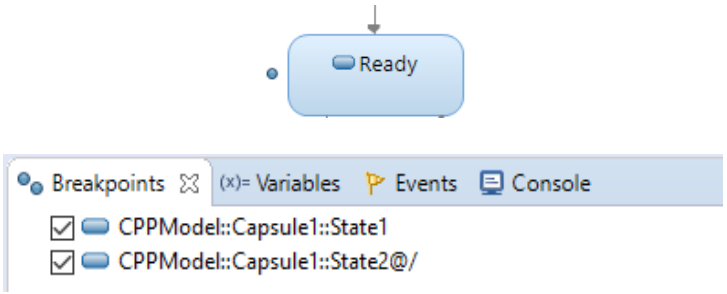
You can set breakpoints on elements in your model. The application will become suspended when a breakpoint is hit, i.e. when the execution reaches an element with a breakpoint. The following types of model elements can have breakpoints:

- **State**
  The breakpoint is hit when the state becomes active (just after its entry code has run)
- **Transition**
  The breakpoint is hit when the transition is about to be triggered (just before its effect code runs)
- **Port**
  The breakpoint is hit when an event is received on the port, before it triggers a transition

There are two kinds of breakpoints: **type-wide breakpoints** and **instance breakpoints**.

A type-wide breakpoint applies for all instances of the capsule that owns the element it is set for. You can set a type-wide breakpoint either in the Project Explorer or in an instance or a regular diagram that shows the element. Instance breakpoints apply for a particular capsule instance only. Therefore you have to set an instance breakpoint from the context of an instance diagram when the debug session has already been started.

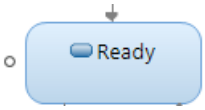
To set a breakpoint for an element right-click on the element and perform the command **Toggle Breakpoint** (or **Toggle Instance Breakpoint**) in its context menu. The breakpoint is shown in the diagram with a small blue ball, next to the element. For example:



Breakpoints are also shown in the Breakpoints view. For each breakpoint the qualified name of the corresponding element is shown, and for instance breakpoints the capsule instance is appended after the @ sign.

From there you can navigate by double-click to the element on which the breakpoint is set. For type-wide breakpoints the element will be selected in a regular diagram. For instance breakpoints, the element will instead be selected in an instance diagram, if a debug session is active. Otherwise it will be selected in a regular diagram.

In the Breakpoints view you can unmark the checkbox to disable a breakpoint. Disabling a breakpoint has the same effect as removing it, but is more convenient in case you later want to enable it again. Disabled breakpoints are shown in diagrams using a hollow blue ball.

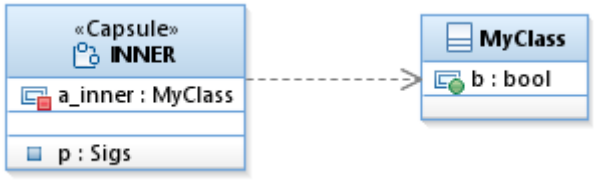


The Breakpoints view is provided by Eclipse and is used by all debuggers, including the C++ debugger. Refer to the Eclipse documentation for more information about the features provided by the Breakpoints view that are the same regardless of which debugger that is used.

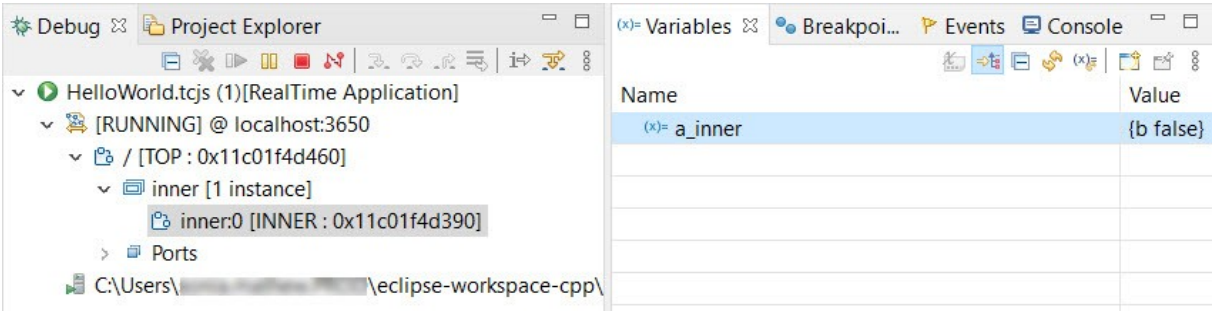
Variables View

Runtime values for capsule attributes are shown in the Variables view when you select a capsule instance in the Debug view.

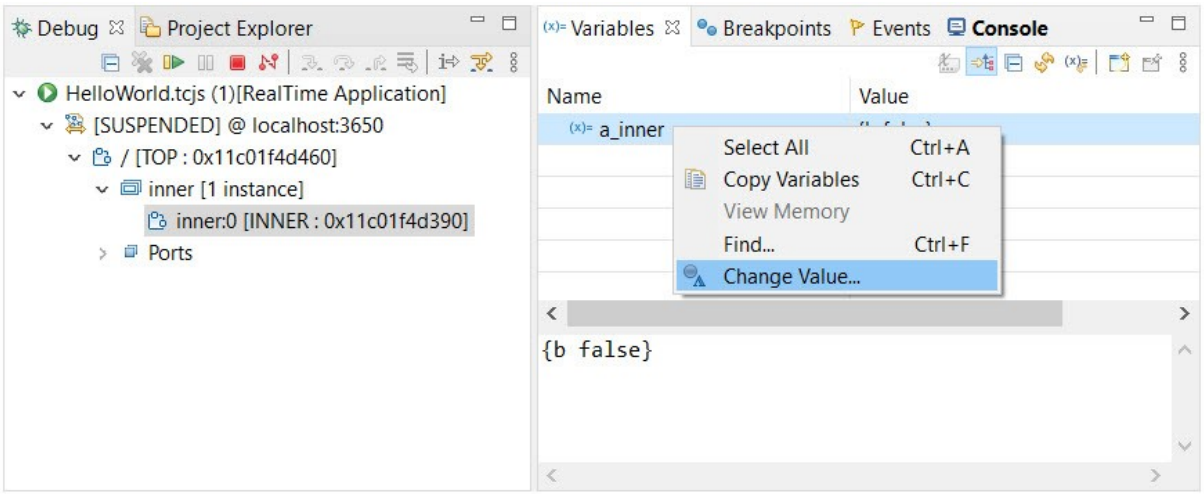
An attribute value is encoded to text using the type descriptor encode function of the attribute's type. As an example, assume we have a capsule with an attribute "a_inner" that is typed by a class MyClass.



When the application is debugged, the Variables view shows the value of this attribute when an instance of the INNER capsule is selected in the Debug view.



You can edit the attribute value by right-clicking on the line in the Variables view only when the application is suspended, and choosing **Change Value** from the context menu. Alternatively, you can directly edit the value in the ”Value” column. Note that the value also is shown in the text area at the bottom of the Variables view. This text area is only for looking at attribute values, and you should not edit the value there.

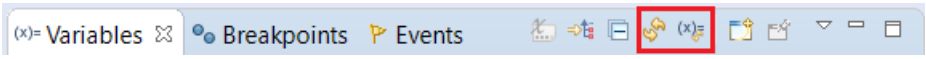


The new value you set for an attribute value is decoded using the type descriptor decode function of the attribute’s type. If decoding fails (for example because you entered a value of an incorrect type) the attribute value is not changed.

The model debugger can show and edit the values of all attributes typed by a type that has a type descriptor, regardless of the visibility of the attribute in the model. If you have attributes typed by complex types, and you find that the default text encoding is hard to read and edit, then it can be a good idea to write your own type descriptor functions (encode and decode) to

simplify viewing and editing the values of those attributes while debugging. Refer to the article in online help at *Model RealTime User's Guide – Articles – Modeling Realtime Applications – Writing a type descriptor* for more details on this topic.

The Variables view shows current values of attributes each time the application gets suspended or when another capsule instance is selected in the Debug view. You can also force a refresh of the Variables view when the application is running by clicking the **Refresh** button in the view tool bar. If you want to automatically refresh the view each time an attribute gets a new value, you can click the **Toggle Monitor Variables** toggle button in the tool bar. This can be useful in order to do a live monitoring of the value of an attribute, as the application is running. However, in this mode the application will run somewhat slower due to the frequent notifications that have to be sent from the running application.



Events View

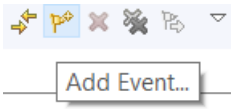
While you are debugging an application using the model debugger, it's often useful to interact with it by manually sending events to a capsule port.

- During a local debug session the manually sent event can emulate what would happen if the application receives that event from an external component when it is deployed in the target environment.
- You can trigger unusual error scenarios by manually sending events in order to debug how the application behaves in those situations.
- You can send a series of events in order to cause a particular capsule instance to transition to an interesting state in its state machine, from where you want to debug its behavior.
- You can send the timeout event to a port typed by the Timing protocol to investigate what will happen when the timer expires.

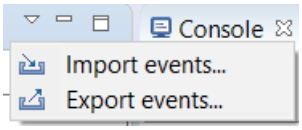
You can send events to any port that is visible in the Debug view, not only to the service ports of a capsule. The only requirement is that the port is typed by a protocol that contains at least one incoming event (or outgoing event in case the port is conjugated). This includes also ports that are typed by protocols from the RT Services Library, for example the Timing protocol.

Events are sent from the Events view. Before you can send any events you must populate the Events view with the events that you want to send. This can be done in a few different ways:

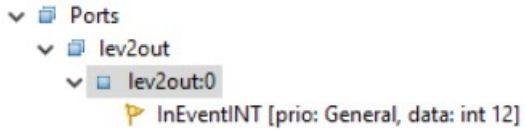
- You can use the command **Populate Events View** that is available in the context menu of ports shown in the Debug view or the Project Explorer. It will add to the Events view all events that can be sent to that particular port.
- You can drag an event from the Project Explorer and drop it in the Events view.
- You can use the **Add Event** command that is available in the Events view toolbar to open a dialog that lets you search for the event to add to the Events view.



- You can import events into the Events view which you previously have exported. Use the commands **Export events** and **Import events** that are available in the view menu of the Events view.



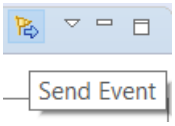
To send an event you can simply drag it from the Events view and drop it onto a port node in the Debug view. If the port is replicated (i.e. has non-single multiplicity) you can drop it on a specific port instance node shown below the port node. If the application is suspended you will see the sent event appear under the port node in the Debug view. For example:



The event will then be dispatched when you resume the application or perform Step Event. If the application was already running when you sent the event, it will immediately be dispatched.

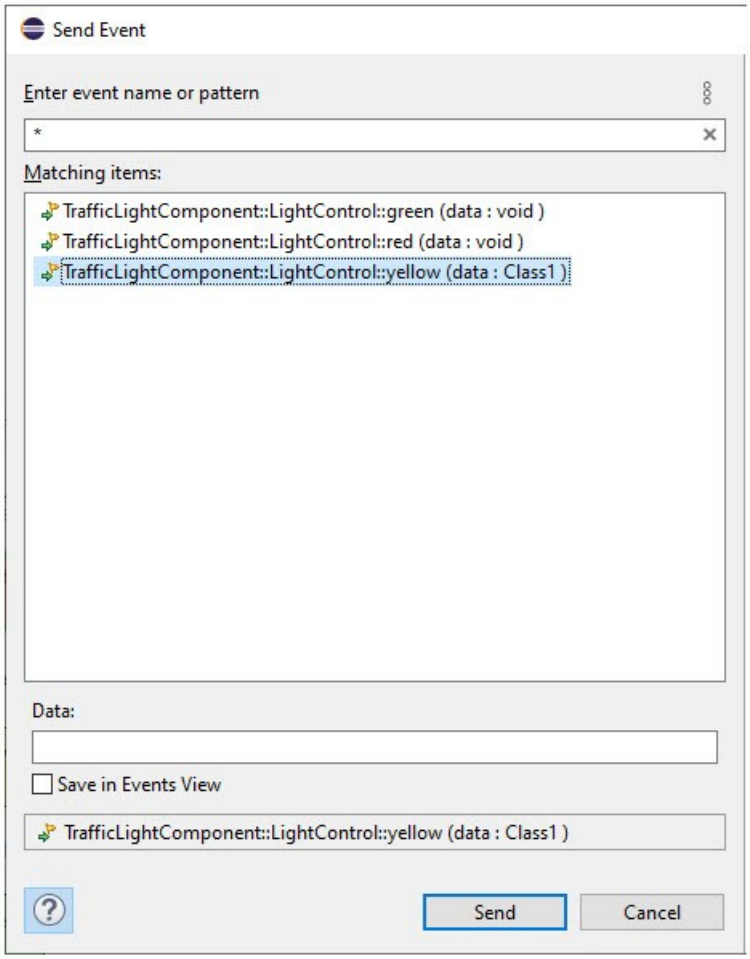
An alternative way to send an event is to first select the receiver port node in the Debug view and then do one of the following:

- Select the event to send in the Events view, and then perform the command **Send Event** which is available both in the toolbar of the Events view and in the context menu of the selected event.

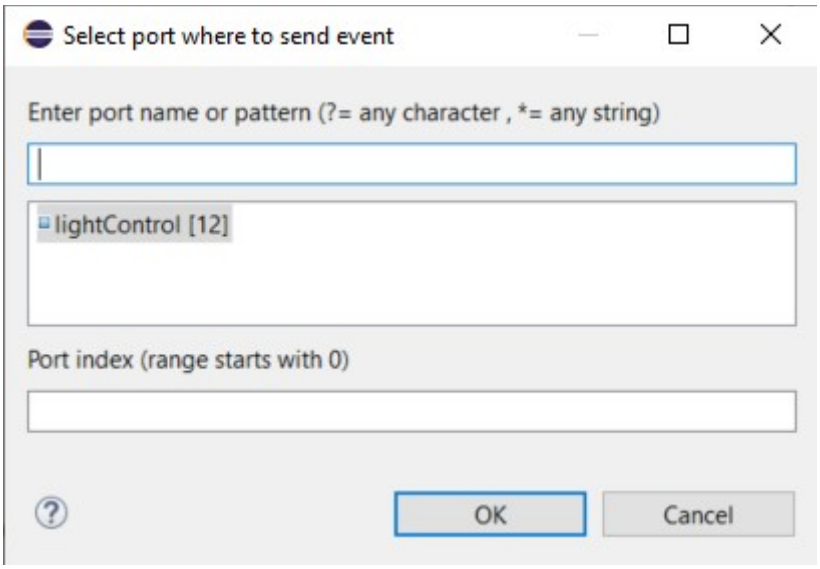


- Right-click on the port node in the Debug view and perform the command **Send Event** in its context menu. This command does not look at what is selected in the Events view, but instead opens a dialog that lets you select an event to send to the port. The dialog lists all events defined in the protocol that types the port. If the port is not conjugated, select an in-event. If the port is conjugated, select instead an out-event. The event parameter type is shown for each event after its name (void if there is no event data). If you select an event with a data parameter, the data field becomes enabled, and you then must provide a data value that matches the event parameter type. The data value must be given on the format that is expected by the ASCII decoder. Unless a custom decode function has been written for the event parameter type, the data value must conform to the rules listed in the documentation.
  Some common examples are:
  - 5, -14 : Integer value
  - true, false : Boolean value
  - 3.14 : Float value
  - "hello!" String value (don't forget the enclosing double quotes)
  - 'a' Char value (don't forget the enclosing single quotes)

- 0, 4 : Enum literal value (corresponds to the declaration index of the enum literal; 0 for the first literal)
- {x 10, y 15} : Struct or class value (with fields x and y of integer type)



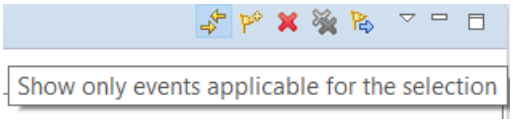
It's also possible to send an event by dragging it from the Events view and then dropping it on a Debug View element that contains one or several ports (such as a capsule instance), or by using the **Send Event** context menu command on such an element. In this case a dialog will appear where you need to select the port to which you want to send the event.



If the port is replicated you can also specify a port index in this dialog (or leave it empty to broadcast the event to all instances of the port). If you dropped an event the dialog will only show the ports that can receive that event, which can help in case there are many ports to choose from. If you instead used the **Send Event** command, another dialog will appear after this one to let you select the event to send on the selected port.

When you send an event that is present in the Events view, the receiver of the event will be stored in the ”Last Sent To” column. When this information has been set for an event you can repeat sending the event to the same receiver by simply double-clicking on the event in the Events view.

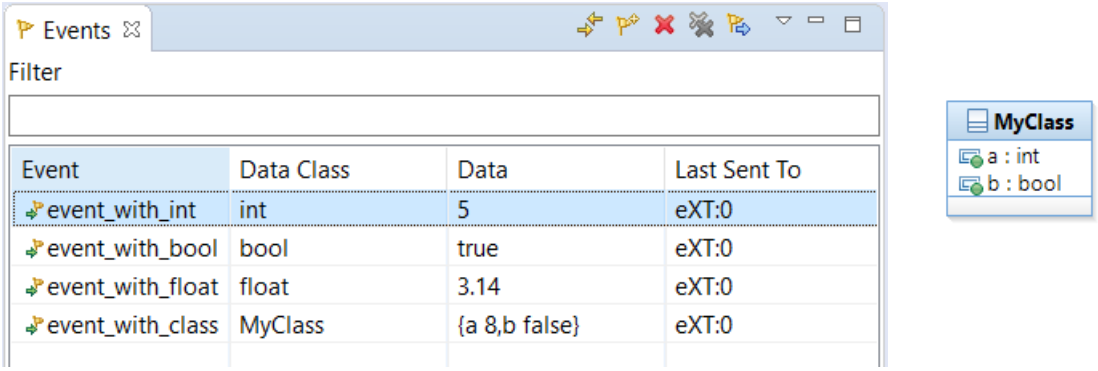
If you have many events in the Events view you can press the toolbar button **Show only events applicable for the selection**. When this toggle button is pressed the Events view will filter the event list to only show those events that can be sent to the port node that you have selected in the Debug view.



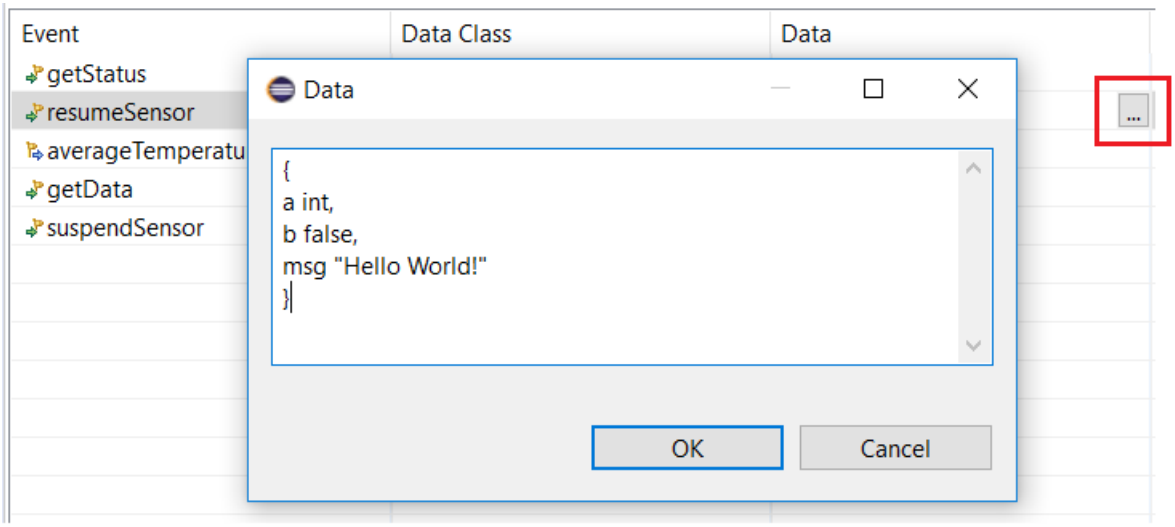
Of course you can also manually filter the list of events by typing some text in the Filter box. The usual wildcard characters (* and ?) can be used.

Specify Event Data

If the event has a data parameter, you must specify the data to send with the event. You should specify the data as a text string on the format that the decode function for the data type expects. Below are some examples of event definitions where the data type uses the default decode function.



For an event with a class or string as data parameter the text in the Data column can be rather long. In this case you can press the browse button in the Data column to open a dialog that lets you edit big data texts more easily.



Hint! If you are unsure about which syntax the decode function for the event data type expects, you can modify your application so that it sends the event with some sample data. Then debug the application and create a trace which will capture the sent event in the trace editor. There you can see how the event data was encoded. The same syntax should be used in the Events view when you want to send that event interactively. See Tracing for more information about how to trace events while debugging an application.

Tracing

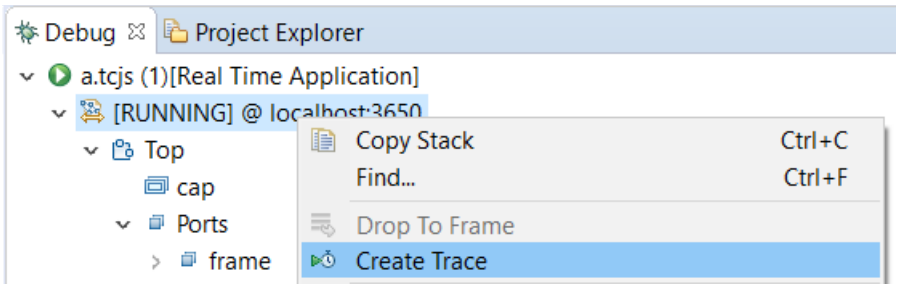
The tracing functionality of Model RealTime allows you to capture traces of what happens in an application during a model debug session. Tracing is a powerful feature that is useful in many situations. Here are a few examples of usecases:

- You can visually inspect so that the application behaves as expected by looking for example at what events that are sent on certain service ports.
- You can see the history of received events that caused a capsule to enter a particular state.

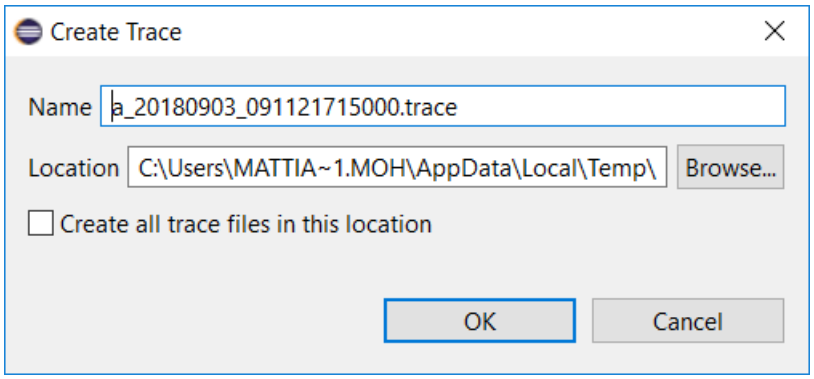
- You can look at how much time is spent in different areas of the application by studying the timestamps in the captured trace.
- You can use a captured trace as a means to run regression tests for an application. Captured traces are text files that a testing script can compare with a previously captured trace.

Create a Trace

To create a new trace, use the context menu command **Create Trace** that is available on most nodes in the Debug view.



This command opens a trace editor that is used both for configuring what trace events to capture, as well as for showing the captured trace events. Before the trace editor opens you need to specify the location of the trace file.



By default trace files are saved in a temporary folder, if you decide to save them. This is because in most cases you will probably not be interested in saving your trace files for future use. Mark the checkbox to avoid getting prompted by the dialog each time a new trace is created. If you, during the trace session, realize that the captured trace is interesting enough to save for the future, you can always use the command **Save As** in the File menu to save it anywhere you like.

In the Create Trace dialog you can also set a name for the trace (i.e. the name of the trace file). The default name is constructed from the name of the TC and the current date and time. Change it to something more meaningful if you plan to save the trace.

When you press OK an empty trace editor will appear. The next step is to configure it to capture the trace events you are interested in.

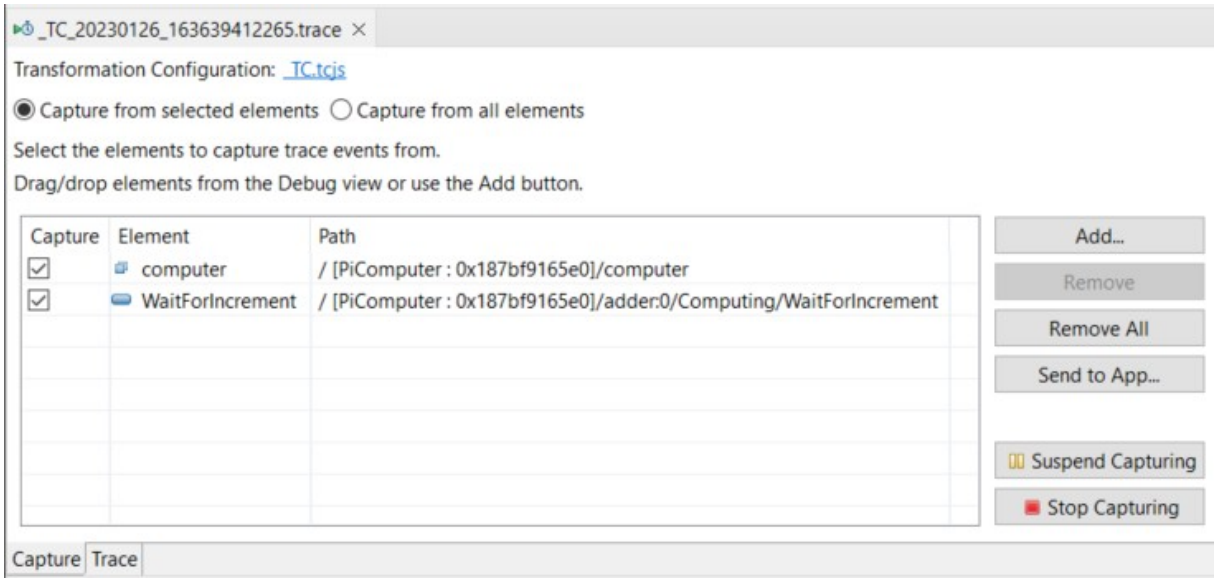
Configure What to Trace

Use the **Capture** tab in the trace editor to configure what trace events to include in the trace. If you created the trace from the context menu of the Process node in the Debug view, then the trace editor will initially be set-up to capture all trace events coming from the application. This can work for smaller applications, or if you only plan to trace during a limited time of its execution. However, for big applications, or trace sessions that span over a long time, it can lead to a huge number of trace events being captured. This can impact negatively on the performance of Model RealTime (although the tool has some mechanisms also for handling very big traces).

If you instead created the trace from the context menu of a Debug view node that represents a run-time element (capsule instance or port instance), then the trace editor will be initially set-up to include only the trace events produced by that particular element. You can add additional elements to trace by either dragging them from the Debug view and dropping them into the table, or by pressing the **Add** button and then browse to the element to trace. Note that some elements, such as states, are not visible in the Debug view, so if you want to trace them you have to add them using the latter approach.

When you add a capsule instance to the list, it's a shorthand for tracing all the ports and states of that capsule instance, as well as the capsule instance itself and capsule instances it contains in its parts, directly or indirectly. If this leads to too many trace events you can be more specific and for example only add some states or ports to the Capture tab.

Here is what the Capture tab may look like for a trace session where we want to trace only some elements. This trace will capture all events that are either sent to or received on the port “computer”. It will also capture when a certain state “WaitForIncrement” becomes active.

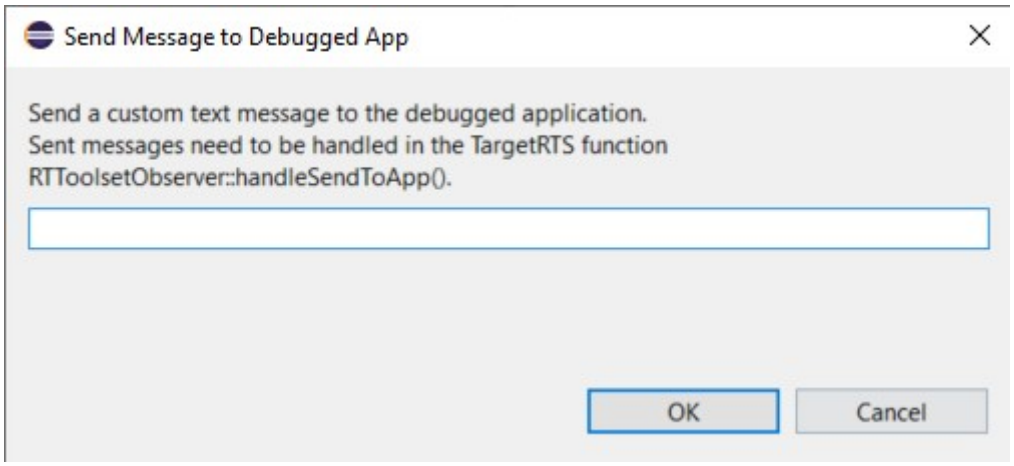


Sub states are qualified with all container composite states, and capsule instances are qualified with all container capsule instances. A slash (/) is used as separator in these state and instance paths.

Note that you can change the configuration of what to trace at any time, also while the application is running and trace events arrive to the trace editor. However, usually it's best to only change the trace configuration when the application is suspended.

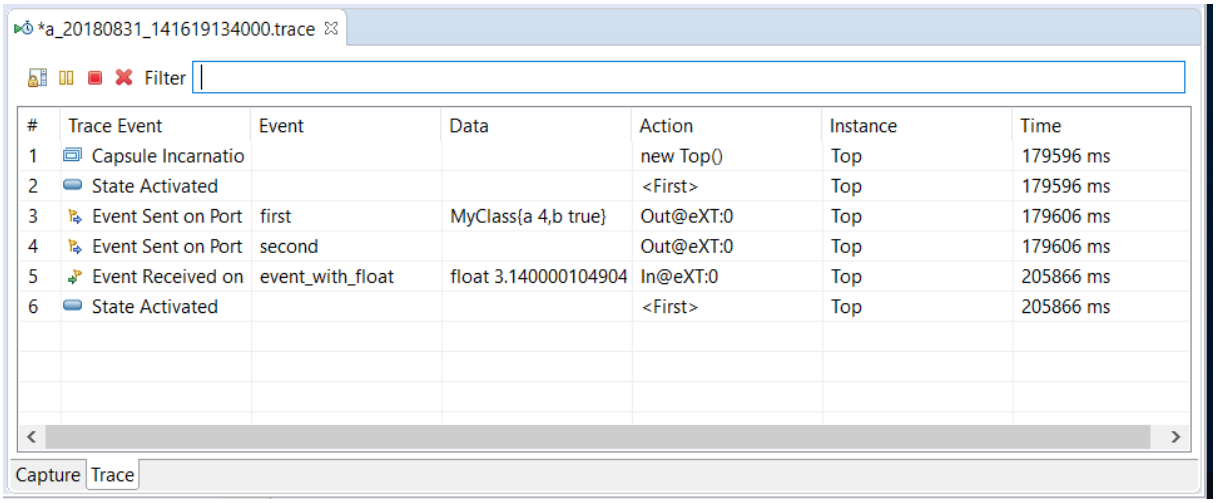
You can temporarily suspend the capturing of trace events. Press the **Suspend Capturing** button to do this. When you later want to resume the trace session again, press **Resume Capturing**. When you don't want to capture any more trace events, press **Stop Capturing**. The trace editor then enters a mode where it cannot capture any further trace events. The Capture tab then becomes disabled, but you can still use the Trace tab for looking at the captured trace.

As can be seen in the picture above the Capture tab also contains a button called **Send to App**. You can use it for sending an arbitrary command (in the form of a text string) to the debugged application. You need to implement the handling of the command in a custom version of the TargetRTS (in the function RTToolsetObserver::handleSendToApp). The default implementation handles a couple of built-in, internally used, commands (starting with the prefix "rt"). For example, you can implement commands that perform certain application-specific tracing by writing information to the console or a log file.



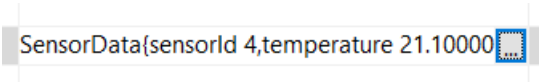
View Captured Trace

The trace editor's **Trace** tab shows all trace events that have been captured from the debugged application. The trace events are shown in the order in which they arrived and are numbered to make them easier to refer to. Here is an example of a trace that contains the most common types of trace events:



The meaning of the different columns in the trace table is explained below:

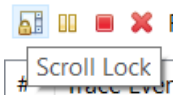
- **Trace Event**
  The kind of trace event. This describes what the application was doing at the point in time when this trace event occurred. Depending on the type of trace event, different information is shown in the other columns.
- **Event**
  This is used for the trace events ”Event Sent on Port” and ”Event Received on Port”. It specifies which event that was sent or received on the port. It's also used for the "initialize" event that is sent to a newly incarnated capsule instance (which happens after the "Capsule Incarnated" trace event).
- **Data**
  This is also only used for trace events related to sending or receiving events on ports, and the "initialize" event. It shows the data carried with the sent or received event. The data is encoded to text using the type descriptor encode function of the data type. If the text is too long to fit in the table cell, you can use the browse button that appears when you click on it, to view the text in a dialog instead.



- **Action**
  Specifies more details about what happened in the application. The format of this text depends on the kind of trace event. For "Capsule Incarnated" it shows the name of the incarnated capsule. For "State Activated" it shows the name of the activated state. And for trace events related to sending or receiving an event, it shows the port instance where the event was sent or received.
- **Instance**
  Specifies the capsule instance in the context of which the trace event occurred.
- **Time**
  A timestamp indicating when the trace event was received. It shows the number of milliseconds since the application was launched.

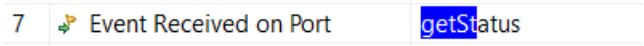
Buttons for suspending or resuming the trace session are available in the upper left corner in the Trace tab. They work the same way as the corresponding buttons in the Capture tab. In addition there is a button for removing all captured trace events.

If you want to start looking at captured trace events while the application is still running, the **Scroll Lock** button can be useful. Press it to avoid that the trace editor scrolls to the bottom each time a new trace event arrives.



Searching and Filtering a Trace

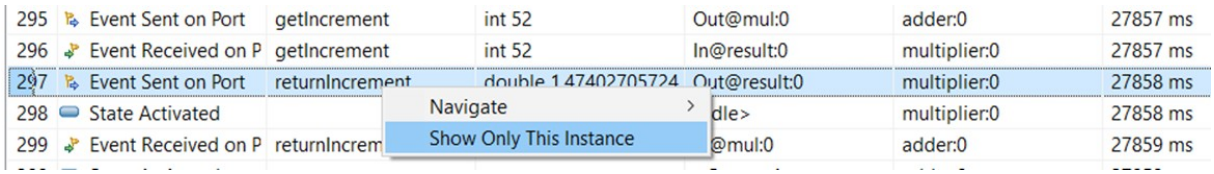
Working with a captured trace usually involves lots of searching, to find particular interesting trace events. The trace editor supports incremental search in a similar way as how it works in for example the CDT editor. Press Ctrl+J to start incremental search, and then type a few characters. The first matching text in the trace will be highlighted as you type:



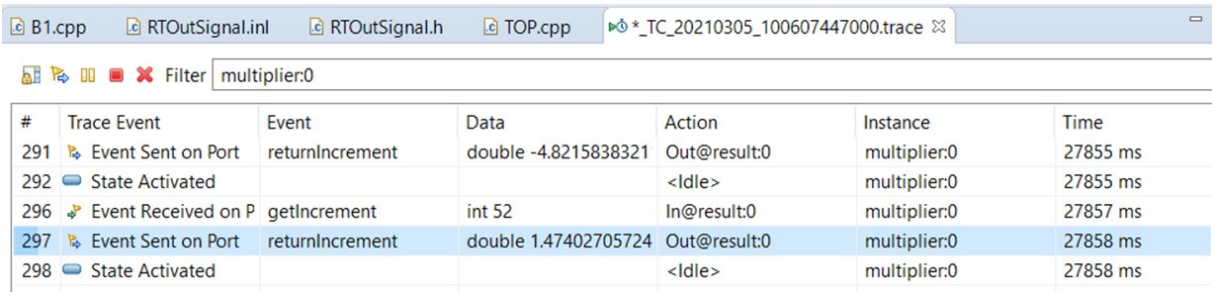
Press Ctrl+J again to go to the next matching text. Press Ctrl+Shift+J to go backwards and select the previously matching text.
You can also search in a trace using the regular Find command (Ctrl + F).

An alternative to searching can be to filter the list of events to only show those that match a certain pattern. Type the pattern in the Filter box. It may contain the usual wildcard characters (* or ?).

A very common filter is to only show trace events related to one particular capsule instance. Applying this filter makes it easy to see what state a capsule instance was in when it sent or received an event. Since this filter is so common there is a context menu command **Show Only This Instance** for applying it. For example, assume we have the following selected trace event showing that a "multiplier:0" instance sent an event on a port:



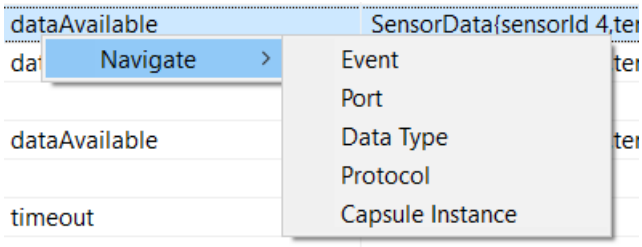
After filtering to only show the "multiplier:0" instance it becomes easy to see that it's active state was "Idle" when this event was sent:



To see all trace events again just remove the text from the Filter box.

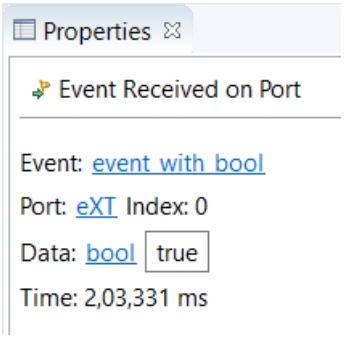
Navigating from Trace Events

You can navigate from a trace event to the corresponding model element that caused it. Right-click on a row in the trace table and use the **Navigate** context menu. It has a sub menu item for each possible element to which you can navigate from the trace event. Navigation is supported both to the definitions of the model elements (in the Project Explorer) and to run-time instances (in the Debug view). The latter is of course only possible as long as the debug session has not been terminated.



View Trace Event Details

When you select a trace event in the Trace editor, details about the trace event are shown in the Properties view. From there you can also navigate to model elements, such as the event or its data type, by clicking on the hyperlinks.



Managing Large Traces

Traces can become very large, especially if you capture trace events from all elements and/or let the trace session run for a long time. To manage large traces the trace editor splits captured trace events into pages, with at most 1000 trace events per page. This avoids performance problems in the trace editor that would arise if all trace events were shown at the same time.

Click the hyperlinks above the trace table to navigate between pages.