APS View

QNX Tool Suite | Integrated Development Environment User's Guide | Developer | Setup

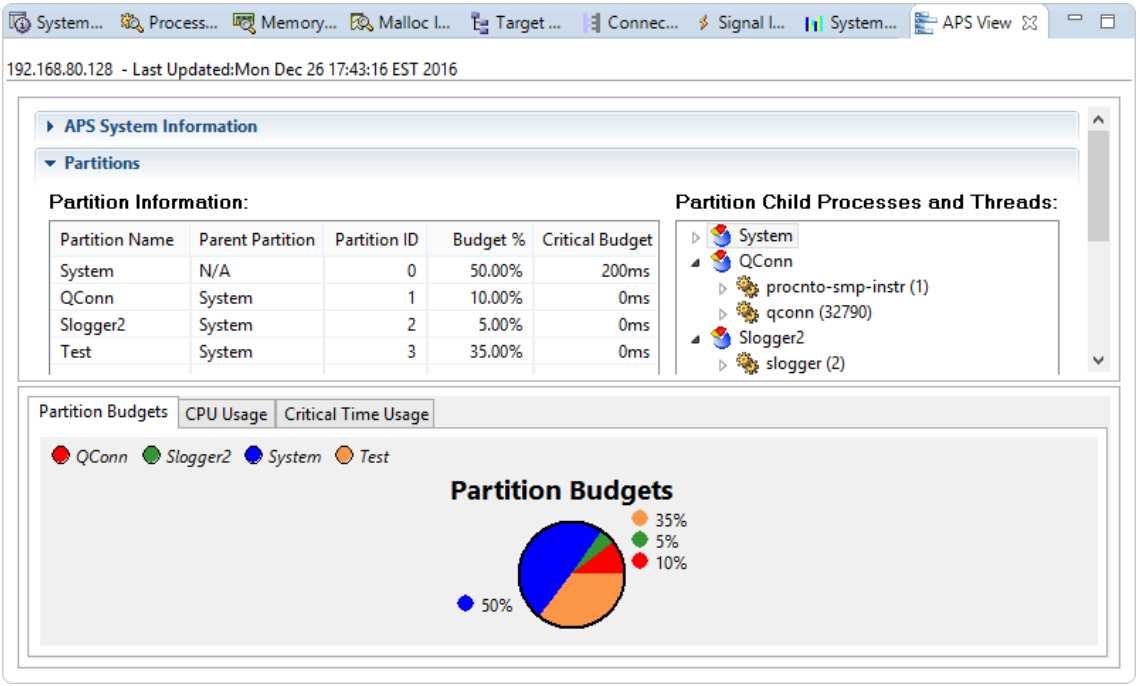
The **APS View** displays information about the adaptive partitioning scheduling (APS) on a target.

NOTE:
QNX OS 8.0 does not support adaptive partitioning; if your target is based on 8.0, then the IDE won't receive any APS data and this pane isn't meaningful. For targets based on an earlier OS version, this pane is meaningful only when the target uses adaptive partitioning.

For more information about adaptive partitioning, see the following content in the user documentation for the OS version that your target is based on:

- the Adaptive Partitioning chapter of the *System Architecture* guide
- the *Adaptive Partitioning User's Guide*
- the `aps` entry in the *Utilities Reference*

To configure APS for a connected target, you must use the Target Navigator. The **APS View** shows APS-related statistics and settings in multiple panels within its top pane; the subsections that follow describe these panels. The bottom pane displays graphs of the partition budgets and CPU usage.



APS System Information

This panel displays the following APS state fields:

- current number of partitions
- partition, process, and thread IDs where the last bankruptcy (exhaustion of critical budget) occurred
- average number of computational cycles executed per millisecond
- *averaging windows*, which are time intervals used by the scheduler to calculate CPU usage
- APS security flags in effect, which determine who can add partitions and modify their attributes
- relative and overall times that the system was idle for longer than the second and third averaging windows (by default, these are 1 and 10 seconds)

NOTE:
When you add or modify partitions using the *SchedCtl()* C library function or the `aps` utility, the changes are reflected in this and all other panels when the view is refreshed.

Partitions

In this panel, the **Partition Information** table shows these details for each partition:

- partition name

- parent partition
- partition ID
- budget, which is the percentage of CPU time guaranteed for the partition
- critical budget, which is the time that critical threads can run over budget

On the right, the **Partition Child Processes and Threads** area lists the processes and threads assigned to each partition. The list is interactive, allowing you to move processes and threads between partitions by dragging and dropping. Moving one of these items might cause others to be moved as well.

Partition Statistics

This panel displays CPU usage statistics related to the averaging windows. The table shows the relative CPU times during which threads in each partition ran for longer than the second and third windows. It also shows the total CPU times that all or just critical thread runtimes exceeded those two windows.

By default, the second and third averaging windows are 1 and 10 seconds, but you can change them by setting a new first window, through *SchedCtl()* or `aps`. The second window is set to 10 times this duration and the third window to 100 times.

APS Bankruptcy

In this panel, the **Bankruptcy Information** table lists the IDs of each partition's notification process and thread (which handle the event emitted when the partition exhausts its critical budget and thus, becomes bankrupt), and the IDs of the process and thread running when the last bankruptcy occurred.

On the right, the **Bankruptcy Flags** area lists the system's bankruptcy settings, which determine how it reacts to a bankruptcy. You can change the bankruptcy settings through *SchedCtl()* or `aps`.

Partition and CPU usage graphs

The **APS View** contains the following graphs in multiple tabs:

Partition Budgets
: This piechart shows the partition budgets, which are the percentages of CPU time assigned to the partitions.

CPU Usage
: This line chart shows recent CPU usage for each partition.

Critical Time Usage
: This line chart shows critical time usage for each partition, which refers to the time (in milliseconds) that critical threads within a partition ran over budget.

Page updated: August 11, 2025

About This Guide

QNX Tool Suite | Integrated Development Environment User's Guide | Developer | Setup

This *User's Guide* describes version 8.0 of the Integrated Development Environment (IDE) that's part of the QNX Tool Suite.

The guide introduces you to the QNX Momentics IDE by explaining the QNX development environment and how to build, run, and debug your QNX OS applications and systems. It then describes scenarios for analyzing memory usage and finding errors in applications.

This guide assumes the following:

- On your host, you've installed the QNX Software Development Platform (SDP), which includes the QNX Tool Suite.
- You're familiar with the *System Architecture* guide of QNX OS.
- You can write code in C or C++.
- You can edit makefiles and in particular, add compiling and linking options when needed.
- You're comfortable using the Eclipse platform.

The following table may help you find information quickly:

To:	Go to:
Start the IDE and set the workspace location	Starting the IDE
Learn about the differences between QNX Momentics IDE and Eclipse and where to find Eclipse documentation	Get to know Eclipse
Create a target connection so you can launch applications on a remote machine	Creating a target connection
Develop and run an application	Developing Projects with the IDE
Find and fix problems in a program, using GDB and other debugging tools	Debugging Applications
Write and run unit tests and measure their code coverage	Unit Testing
Monitor memory usage, analyze heap activity, and find memory corruption and leaks	Analyzing Memory Usage and Finding Memory Problems
Monitor performance, profile applications, and investigate performance bottlenecks	Analyzing Performance
Run system-wide diagnostics tools to assess system behavior	Analyzing System Behavior
Build images for QNX OS target systems	Building QNX OS Images
Learn about the QNX perspectives	QNX perspectives
See the full list of launch configuration properties	QNX launch configuration properties
Understand the integrated development aid tools	Integrated tools

Page updated: August 11, 2025

Adding libraries

Our sample application doesn't use libraries but if you want to build an application that does, you need to link those libraries to your application. The way you do this in the IDE depends on the type of makefile you're using.

If your project uses standard makefiles, you must edit them and add libraries by setting the appropriate flags (command options) in the compiling and linking commands. For details on the required command options, see the `q++, qcc` entry in the *Utilities Reference*. If your project uses recursive makefiles, you can configure library linking through the QNX C/C++ Project properties.

NOTE:
When developing libraries, you can link other libraries into their projects. This is done the same way as with application projects, for both standard and recursive makefiles.

Page updated: August 11, 2025

Analyzing heap memory usage

The IDE offers several ways to analyze an application's heap memory usage. You can view realtime heap statistics, run analysis tools to track heap activity, and use the **libc** API to read heap information.

Some tools provide high-level data such as total, used, and free heap space and have low overhead and setup cost. Other tools provide detailed heap information such as allocation counts for specific block size ranges but require more configuration and impose more overhead. Generally, you should try using the heap analysis tools in the following order:

1. **System Information**

   In the QNX System Information perspective, you can monitor the heap usage for a particular process through the **Malloc Information** view. Using the debugger, you can stop execution at interesting places in the code and see which pointers have new values (addresses). If you then switch to the **Memory Information** view, you can see if the segments containing those addresses have grown.

   This is the best tool to use first because you can switch to this perspective at any time to see the statistics. You don't need to reconfigure, recompile, or relaunch your application, and there's no application overhead because the statistics come from the general-purpose process-level allocator.

2. **Memory Analysis**

   This tool displays data reported by the debug allocation library (**librcheck**) as the program runs, providing you with realtime statistics about a process's dynamic memory events and changes in its heap usage.

   This tool has an easy setup and imposes less overhead than other analysis tools but requires you to relaunch the application with the tool enabled.

3. **Valgrind Massif**

   Valgrind Massif is designed for heap analysis. This tool can tell you more than Memory Analysis; for example, it reports the peak in heap usage and provides better backtracing details.

   You don't need to recompile the application but you do need to configure Massif through the launch configuration. A drawback to using Valgrind is that it reports the results only when the program terminates, so you can't view memory usage in real time. Also, Massif imposes significantly more overhead than Memory Analysis.

4. **Application Profiler**

   This tool parses the profiling data generated when an instrumented binary is run. In the launch configuration, you can specify heap memory as the profiling metric to make the IDE display the program's heap usage by function.

   Profiling has a higher setup cost than other analysis strategies because you must compile the project with the correct flags and relaunch it. The benefit is that the Application Profiler presents the heap data in an easy-to-read format and lets you compare the results from different sessions.

5. **The libc allocator API**

   If you're comfortable using the **libc** allocator API, you can read the fields in the `mallinfo` structure to learn the total allocated space, total free space, number of arenas, and more.

   To use this API effectively, though, you must know the code well enough to determine suitable places for querying and outputting the memory allocation statistics. Also, you must rebuild and relaunch the application to start outputting them.

You must run a binary built with debug information so the active tool can match binary instructions with lines of code and display the symbols (i.e., backtrace) in the results. To see symbols for shared libraries, your host machine must store debug versions of these libraries. Any binary with debug information has a bug icon () next to its name in the Project Explorer. The binary selected for running when you launch a project depends on the launch configuration settings.

Page updated: August 11, 2025

Importing results from memory-analyzing sessions

The IDE can import session data generated by a memory-analyzing tool while an application ran. This allows you to view the results of previous analyses, including those shared by other team members or those from sessions that you ran outside of the IDE.

For some development environments, it might not be possible to start an analysis session from the IDE. For example, if the host and target machines aren't connected to the same network, or if you don't have **qconn** or a similar target service to forward session data to the host.

In these cases, you can run the application with an analysis tool from the command line, copy the data files generated during the session to your host, then import them into the IDE to view the application's memory usage or corruption errors and leaks. This workflow is known as a *postmortem analysis*.

The sections that follow explain how to import session data for each tool.

Page updated: August 11, 2025

Analyzing Memory Usage and Finding Memory Problems

QNX Tool Suite | Integrated Development Environment User's Guide | Developer | Setup

QNX Momentics IDE lets you run Memory Analysis and several Valgrind (pronounced “val-grinned”) tools to find memory problems such as leaks and corruption, and to measure memory usage of programs. It also provides the QNX System Information perspective so you can learn which processes are consuming memory or other resources excessively.

The analysis tools support these four tasks:

- monitoring memory and resource consumption — you can view realtime data about memory usage for a system or a single process and about resource usage for a process; this feature helps you determine which processes should be further analyzed or optimized
- analyzing heap memory usage — you can see which areas of a program use the most dynamic (heap) memory, to learn which sections of code need to be optimized to improve performance
- finding memory corruption — you can find and fix the bad memory operations that cause a program to crash or behave improperly
- finding memory leaks — you can find and fix memory leaks, to improve an application's long-term stability and performance

The System Information data and the results from the runtime analysis tools can reveal many specific memory problems that you can fix as well as usage trends that suggest how you might optimize an application. You can export data from an analysis session to share results with other team members, who can then import them into their IDE. The import feature also supports postmortem analysis, in which you view analysis results for an application that was run outside of the IDE.

Page updated: August 11, 2025

Exporting results from memory-analyzing sessions

The IDE can export session data generated by a memory-analyzing tool while an application ran. This allows you to share analysis results or view them outside of the IDE.

When an IDE-launched analysis session ends (i.e., the program terminates), all of the session data have been stored in the workspace. At this point, you can export the session. The export feature is handy because it writes the analysis results in a format that can easily be imported into the IDE again; for example, into the IDE of another team member with whom you share the results.

You can view the results outside of the IDE. Memory Analysis lets you export event data as CSV files, which you can then view in Excel. The Valgrind tools generate text-based data, which you can view in any basic editor after exporting the text files.

The sections that follow explain how to export session data for each tool.

Page updated: August 11, 2025

Importing performance analysis results

The IDE can import session data generated by a performance-measuring tool while an application ran. This allows you to view the results of previous analyses, including those shared by other team members or those from sessions that you ran outside of the IDE.

For some development environments, it might not be possible to start an analysis session from the IDE. For example, if the host and target machines aren't connected to the same network, or if you don't have **qconn** or a similar target service to forward session data to the host.

In these cases, you can run an instrumented binary from the command line, copy the profiling results to your host, then import them into the IDE to view the application's call counts and function runtimes. This workflow is known as a *postmortem analysis*.

You can also create an Application Profiler session by importing a kernel event log (**.kev**) file. The new session contains the profiling results generated by an instrumented application binary that ran during the kernel event trace. Details about importing this type of profiling data as well as data generated by other supported methods are given in the next section.

For Valgrind Cachegrind, you can import a text file containing the results of a previous cache usage analysis by following the procedure given in "Importing Valgrind logs".

Page updated: August 11, 2025

Exporting performance analysis results

The IDE can export session data generated by a performance-analyzing tool while an application ran. This allows you to share analysis results or view them outside of the IDE.

When an IDE-launched profiling session ends (i.e., the program terminates), all of the session data have been stored in the workspace area. At this point, you can export the session. The export feature is handy because it writes the analysis results in a format that can easily be imported into the IDE again; for example, into the IDE of another team member with whom you share the results.

You can view the results outside of the IDE. The Application Profiler lets you export function data as CSV files, which you can then view in Excel. Valgrind Cachegrind generates text-based data, which you can view in any basic editor; this workflow is explained in "Exporting Valgrind logs".

Page updated: August 11, 2025

Analyzing Performance

QNX Tool Suite | Integrated Development Environment User's Guide | Developer | Setup

QNX Momentics IDE works with the Application Profiler and Valgrind (pronounced “val-grinned”) Cachegrind tools, which measure the performance of programs running on QNX targets. It also offers the QNX System Information perspective for viewing performance-related metrics.

NOTE:
The term *performance* can have many meanings. In this guide, it refers to application responsiveness and the speed of operations. Memory consumption, particularly on the heap, is an important part of performance; this topic is covered in the previous chapter. Here, we discuss performance primarily from a speed perspective.

The analysis tools support these three tasks:

- monitoring performance — you can examine realtime performance statistics for processes and threads to determine which applications need to be further analyzed and optimized
- profiling applications — you can profile an application to learn how often its individual functions are called and how much time is spent in each function
- investigating performance bottlenecks — you can measure an application's cache misses and heap memory usage, and see how it interacts with other processes to determine why it runs slowly

You can export performance analysis data to share results with other team members, who can then import them into their IDE installations. The import feature also supports postmortem analysis, in which you view analysis results for an application that was run outside of the IDE.

Page updated: August 11, 2025

Analyzing System Behavior

Many IDE tools allow you to find errors and optimize applications, but to ensure that your embedded system behaves well, you need to use system-wide diagnostic tools. The QNX System Information and QNX System Profiler tools help you understand process and thread interaction on the target, reduce application and system startup times, and debug deadlock and improper CPU usage levels.

The system analysis tools support these four tasks:

- finding bad interactions — through System Information, you can detect deadlocked threads and odd connections between components, and see how a process handles certain signals
- verifying correct thread priorities — through System Information, you can monitor threads to verify that they run at the correct priorities
- analyzing kernel activity — you can run a kernel event trace through the IDE, then examine the trace results in the System Profiler to see how CPU resources are distributed and how threads are scheduled
- optimizing system performance — based on kernel event trace results, you can make individual applications or the entire system boot faster

From the System Profiler editor, you can extract any profiling data captured during the trace period into a new Application Profiler session.

Page updated: August 11, 2025

Profiling system activity with function instrumentation enabled

QNX Tool Suite | Integrated Development Environment User's Guide | Developer | Setup

You can perform a kernel event trace while running applications with function instrumentation enabled. Combining application and system profiling lets you log the events generated by the instrumentation code to the kernel event trace log. These events provide function entrance and exit times and thread call stacks.

This workflow is handy when you suspect that other processes are impacting an application's performance at startup because the kernel event trace tells you how much memory and CPU is consumed by different target processes. It's also useful for examining the interaction between a short-running program and other processes. However, any tracing activity should be limited to a few seconds because longer traces produce too much data to be useful.

To profile functions, your source files must be compiled with `-finstrument-functions` and your binaries linked with `-lprofilingS`; for details, see "Enabling function instrumentation".

NOTE:
You can run an application with profiling instrumentation from the command line. However, using the IDE is more convenient because the Application Profiler automates the requesting of data from the instrumented binary and presents the results in an easy-to-read format.

To profile system activity while measuring function runtimes:

1. In the launch bar, expand the Launch Configuration dropdown (which is in the middle) and select the project that you want to profile.
2. In the Launch Target dropdown (on the right), select the target for running your application.
3. In the Launch Mode dropdown (on the left), select Profile.
4. Click the Edit button (⚙) on the right of the Launch Configuration dropdown.
5. In the configuration editor window, click the **Profile** tab on the right to access the Application Profiler controls.
6. Ensure that **Functions Instrumentation** is selected under **Profiling Method** and **System Wide** is selected under **Profiling Scope**.
7. **Optional:** You can customize how the profiling tool behaves through the **Options** and **Control** panels. For details about these fields, see the Application Profiler reference.
8. Check the **System Profiler** box at the bottom of the **Profile** tab to enable kernel event tracing.
9. **Optional:** You can customize how the kernel event trace is done in the UI fields shown for this tool. The **Wait interval** text field lets you set a delay, in milliseconds, for starting the trace after the application is launched. The **Kernel Log configuration** dropdown lets you select the log configuration for controlling the trace. Or, you can modify the selected configuration by clicking **Edit**.
10. Click **OK** to save the configuration changes and close the window.
11. In the launch bar, click the Profile button ( ).

The IDE switches to the QNX Analysis perspective. If necessary, the IDE first builds the binary. Then, it uploads the binary and starts running it on the target. At this time, a new session is created and displayed in the **Analysis Sessions** view. As the application runs, profiling results are sent to the IDE, which stores them in the new session and presents them in the **Execution Time** view.

The kernel event trace also runs and when it finishes, the IDE prompts you to open the log (**.kev**) file. If you choose `Yes`, the IDE switches to the QNX System Profiler perspective and displays the trace results. The application being profiled continues to run (assuming it hasn't terminated on its own) and you can still switch back to the QNX Analysis perspective to examine the profiling results at any time. Note that with function instrumentation, deep and shallow times have distinct meanings.

When viewing the trace results, you can extract any profiling data captured during the trace period into a new Application Profiler session.

NOTE:

You can concurrently profile as many applications as you like and multiple instances of the same application. The results for each execution run appear in their own Application Profiler session, independently of other sessions.

Page updated: August 11, 2025

Attaching Application Profiler to a running process

If you want to start seeing function runtimes and call counts for a running process, you can attach the Application Profiler tool. The IDE then displays the profiling data received from the instrumented binary running on the target.

NOTE:
The application that you want to profile must have been built with profiling instrumentation, either for sampling execution positions or measuring function runtimes, then run from the command line. Also, your host machine must have an IP connection to your target machine.

To attach the Application Profiler to a running process:

1. In the launch bar, expand the Launch Configuration dropdown (which is in the middle) and select the application for which you want to see profiling data.
2. In the Launch Target dropdown (on the right), select the target on which the application is running.
3. In the Launch Mode dropdown (on the left), select Attach.
4. Click the Edit button () on the right of the Launch Configuration dropdown.
5. Enable the Application Profiler in the launch configuration:
    a. Click the **Tools** tab on the right.
    b. Click the Application Profiler radio button.
    c. Ensure that the **Profiling Method** selection matches the instrumentation used by the application.
    d. **Optional:** If needed, configure other tool settings to customize how data are transferred to the IDE and which signals are used to start and stop profiling.
    Based on these settings, the Application Profiler will modify how the application is behaving when it's attached to the application process.
    e. **Optional:** Enable or disable other analysis tools.

    CAUTION:
    Don't enable the debugger when attaching the profiling tool. Stopping at breakpoints and stepping through code makes the function runtimes completely inaccurate. You should therefore ensure the debugger checkbox is unchecked at the top of the **Debug** tab.

    If you chose a **Profiling Scope** of `System Wide`, you must also run the System Profiler by checking this tool's box at the bottom of the **Tools** tab. This makes the IDE perform a kernel event trace while gathering profiling data.
6. When you've finished configuring the tools, click **OK** to save the changes and exit the window.
7. Click the Attach button ().
    The IDE switches to the Debug perspective if the debugger is enabled, or the QNX Analysis perspective if it's not, then opens the **Select Process** popup window. This window lists the processes with the same name as the binary specified in the **Main** tab.
8. Click the process that you want to attach to, then click **OK**.

The IDE attaches the Application Profiler and any other enabled tools to the selected process. In the current perspective, you'll see a new session for storing the debugging or profiling results. Depending on the active tools and their configuration, these results might get updated as the program runs.

In the QNX Analysis perspective, you can see the function runtimes and call counts in the **Execution Time** view. The Application Profiler reference explains how to interpret the results.

Page updated: August 11, 2025

Comparing profiling session results

Typically, profiling is part of an iterative development process in which you make changes to improve efficiency in some application areas based on the latest performance results. The IDE supports this process by letting you easily compare the results of two Application Profiler sessions.

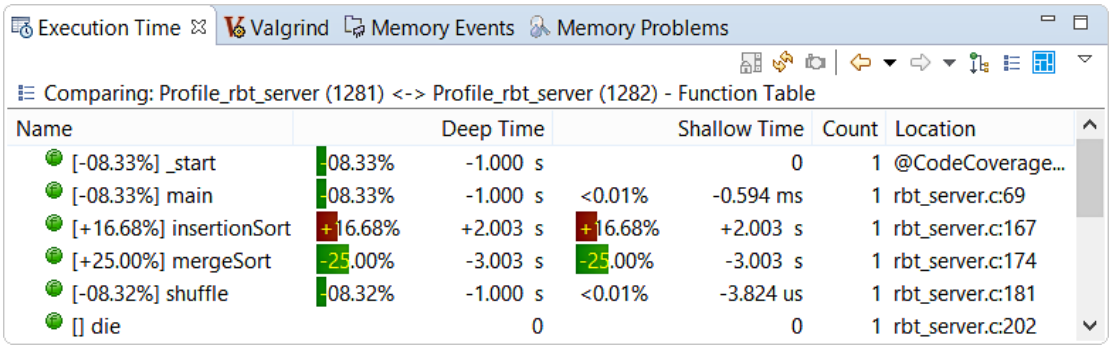
The comparison feature works for both profiling methods—that is, when sampling positions and counting calls or when measuring function runtimes. It's not recommended to compare sessions that used different methods because sampling-based estimates are less precise than function measurements.

Also, a comparison is meaningful only when you profiled the same application with the same workflow but different implementations, based on code changes you made to improve performance.

To compare the results of profiling sessions:

1. In the **Analysis Sessions** view, use multiselect to highlight two Application Profiler sessions.
   The comparison feature works only when exactly two sessions are selected.
2. Right-click and in the context menu, choose **Compare**.

In the **Execution Time** view, the IDE displays the differences in the results from two sessions. The functions table, which you can select by clicking the **Show Table** button ( ) in the upper right corner, is best for seeing these differences:



Full details about the numbers and icons shown in the comparison results are given in the Application Profiler reference. In summary, the **Deep Time** and **Shallow Time** values indicate the increase or decrease in execution time from the first (older) to the second (newer) session. In the program used to generate the results shown above, the *shuffle()* function calls *mergeSort()* during the first session but *insertionSort()* during the second session. This latter operation runs faster, as seen by the decrease in deep time for *main()* (i.e., the whole program) and the time for the insertion sorting function being less than that for merge sorting. Thus, performance has improved with the new sorting method.

Page updated: August 11, 2025

Configuring shared library support

The integrated performance measurement tools produce results that contain statistics for individual functions. To include information about functions from shared libraries, the active tool must have access to library copies with debug symbols and possibly, profiling instrumentation.

To find these debug symbols, the tools search the paths listed in the **Libraries** tab. When you add a library to a project, the IDE normally adds the path with the debug version of the library to the search list. It also adds the library file to the list of files to upload to the target, defined in the **Upload** tab. You can strip the debug information when uploading this file—the IDE just needs to find the debug symbols somewhere on the host when displaying the results.

If the automation fails, though, the user must manually specify where to find the right library files. The way to do this depends on the tool.

NOTE:
If you're seeing line numbers for a shared library but double-clicking its function entries takes you to unexpected code locations, you have a mismatch between the host version of the library that contains debug symbols and the target version that gets loaded by the executable. You should therefore rebuild the application to link in the correct version, and verify that the **Upload shared libraries** list (in the **Upload** tab) contains the right library file.

Application Profiler

If you're not seeing function runtimes and call counts for some shared libraries in the profiling results, you must manually define the library paths using the same steps as with Memory Analysis. Note that the library versions stored at these paths must contain profiling instrumentation. The compiler and linker options required for this are different for obtaining call counts in sampling mode versus measuring precise runtimes.

Valgrind

The Valgrind tools can request debug symbols from a symbol server on the host and use this information in writing the analysis results. Usually, the server is already running so you don't need to do anything to see statistics for library functions in Cachegrind results. However, if this isn't happening for some reason, you can manually start the server or specify target paths for finding the libraries.

Page updated: August 11, 2025

Measuring function runtimes

You can instrument an application binary to report timestamps of function entrances and exits. The Application Profiler can then measure and report the total time spent in each function, allowing you to monitor execution time breakdown based on the application's workflow (i.e., action sequence).

This profiling method has more overhead than sampling execution positions but doesn't require you to gather a statistically significant number of samples, meaning it works on short- or long-running programs. It performs better on one thread because with many threads, the measurement overhead can change the application's behavior.

To profile functions, your source files must be compiled with `-finstrument-functions` and your binaries linked with `-lprofilingS`; for details, see "Enabling function instrumentation".

NOTE:
You can run an application with profiling instrumentation from the command line. However, using the IDE is more convenient because the Application Profiler automates the requesting of data from the instrumented binary and presents the results in an easy-to-read format.

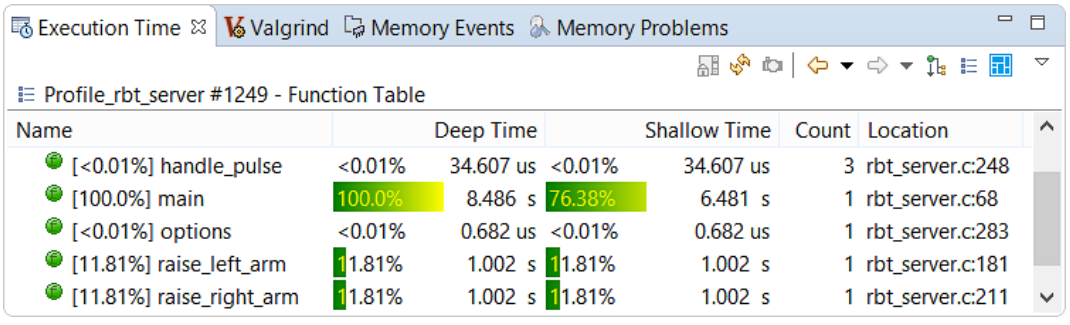
To profile an application by measuring function runtimes:

1. In the launch bar, expand the Launch Configuration dropdown (which is in the middle) and select the project that you want to profile.
2. In the Launch Target dropdown (on the right), select the target for running your application.
3. In the Launch Mode dropdown (on the left), select Profile.
4. Click the Edit button (⚙) on the right of the Launch Configuration dropdown.
5. In the configuration editor window, click the **Profile** tab on the right and the Application Profiler radio button near the top of this tab.
6. Ensure that **Functions Instrumentation** is selected under **Profiling Method** and **Single Application** is selected under **Profiling Scope**.
7. **Optional:** You can customize how the profiling tool behaves through the **Options** and **Control** panels. For details about these fields, see the Application Profiler reference.
8. Click **OK** to save the configuration changes and close the window.
9. In the launch bar, click the Profile button ( ).

The IDE switches to the QNX Analysis perspective. If necessary, the IDE first builds the binary. Then, it uploads the binary and starts running it on the target. At this time, a new session is created and displayed in the **Analysis Sessions** view. As the application runs, profiling results are sent to the IDE, which stores them in the new session and presents them in the **Execution Time** view.

The profiling results include the function runtimes, call counts, and call sites. Initially, function information for all program components is listed. You can expand the **Analysis Sessions** entry and click a component to filter the display. For example, you may want to see only those runtimes for functions in your program code and not in any libraries that it uses, so you can click the application binary.

By default, the **Execution Time** view shows the threads tree, but for single-threaded applications, it's better to click the **Show Table** button ( ) in the upper right corner to list the functions in a table:



You should be able to see statistics for functions in any shared libraries used by your application. If you don't, you must manually define the library paths.

One key point in using function instrumentation is that deep and shallow times have distinct meanings. Deep time is the time spent in a function's own code and in the code of any functions that it calls. Shallow time is the time spent in a function's code and inside any functions that it calls but aren't instrumented themselves. Generally, this is all of the QNX library. Note that shallow time can't be computed for functions still executing. You can change the columns displayed to see nondefault metrics, as explained in the Execution Time reference.

Like the System Information statistics, the profiling results are refreshed every five seconds (by default), so you can closely monitor changes in execution time breakdown as you interact with the application. Double-clicking on any listing with a file and line number shown in the **Location** column opens the source code at that location. The IDE displays a bar graph and the execution time in the left margin.

NOTE:
You can concurrently profile as many applications as you like and multiple instances of the same application. The results for each execution run appear in their own Application Profiler session, independently of other sessions.

Page updated: August 11, 2025

Analyzing heap memory usage with Application Profiler

An application can be instrumented to measure and report its heap size at every function entry and exit. The Application Profiler can then calculate each function's heap size changes and provide a detailed breakdown of heap memory usage.

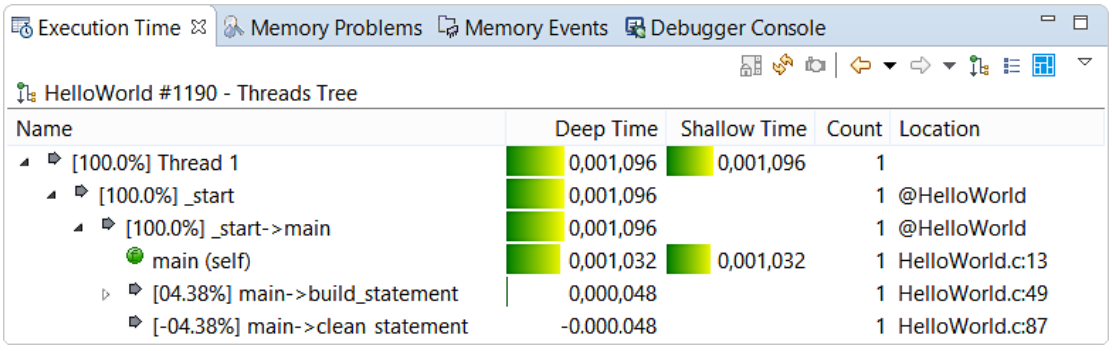
To profile functions, your source files must be compiled with `-finstrument-functions` and your binaries linked with `-lprofilingS`; for details, see "Enabling function instrumentation".

To analyze heap memory usage with the Application Profiler:

1. In the launch bar, expand the Launch Configuration dropdown (which is in the middle) and select the project for which you want to analyze heap usage.
2. In the Launch Target dropdown (on the right), select the target for running your application.
3. In the Launch Mode dropdown (on the left), select Profile.
4. Click the Edit button (⚙) on the right of the Launch Configuration dropdown.
5. In the configuration editor window, click the **Profile** tab on the right and the Application Profiler radio button near the top of this tab.
6. Ensure that **Functions Instrumentation** is selected under **Profiling Method** and **Single Application** is selected under **Profiling Scope**.
7. In the **Profiling Counter** dropdown, select **Memory: Allocated Heap**.
8. **Optional:** You can adjust other profiling settings through the **Options** and **Control** panels.
   For details about these fields, see the Application Profiler reference.
9. Click **OK** to save the configuration changes and close the window.
10. In the launch bar, click the Profile button ( ).

The IDE switches to the QNX Analysis perspective. If necessary, the IDE first builds the binary. Then, it uploads the binary and starts running it on the target. At this time, a new session is created and displayed in the **Analysis Sessions** view. As the application runs, profiling results are sent to the IDE, which stores them in the new session and presents them in the **Execution Time** view.

By default, this view lists the functions by thread:



Here, the **Deep Time** column indicates by how much a function increased heap memory consumption (when the value is positive) or decreased it (when the value is negative). Note that a function may allocate some blocks but free others—the value shown is the net increase or decrease in heap memory. **Shallow Time** (if defined) reports the same metric, so you can ignore this column. **Count** indicates how many times the call site was executed while **Location** provides the source file and line of the call site.

The controls in the upper right corner allow you to display certain columns so you can see non-default metrics and to display the functions in a flat list rather than a thread-based tree, as explained in the Execution Time reference. Also, you can compare the results of two profiling sessions to see the effects of any changes you made between execution runs to improve application efficiency.

NOTE:
You can concurrently profile as many applications as you like and multiple instances of the same application. The results for each execution run appear in their own Application Profiler session, independently of other sessions.

Page updated: August 11, 2025

Profiling applications

QNX Tool Suite | Integrated Development Environment User's Guide | Developer | Setup

From the IDE, you can build an application for profiling and launch it. The Application Profiler then displays, in the QNX Analysis perspective, how much execution time is consumed by different functions.

Profiling applications helps you pinpoint inefficient areas in your code, without having to follow execution line by line. The Application Profiler tool lets you measure precise function runtimes or obtain runtime estimates based on execution position sampling. When measuring runtimes, you can also run a kernel event trace to see when an application's functions are executed relative to other system events.

When you enable profiling for an application or library, the binary that gets built includes debug information. This is so the profiler tool can match addresses in the results with lines of code and display the symbols (i.e., line numbers backtrace). Any binary with debug information has a bug icon () next to its name in the Project Explorer. The binary selected for running when you launch a project depends on the launch configuration settings.

For applications already running, you can attach the Application Profiler tool to start seeing results in the IDE. You can also perform a postmortem analysis, in which you import profiling results from an application that was run from the command line.

Page updated: August 11, 2025

Sampling execution position and counting function calls

Any application launched through the IDE reports its execution position at regular intervals. Using the position samples, the Application Profiler can estimate function runtimes. You can also instrument a binary to make it report every function caller, so you can see call counts in the profiling results.

This profiling method has less overhead than measuring function runtimes but isn't perfect because a function can be called many times between sampling intervals and you won't see any references. Also, you can't use this method on timer-synchronized programs because the results won't be accurate.

For accurate results, you must obtain a statistically significant number of samples, so this method works best on long-running programs that spend most of their time executing code. For instance, if a server process is mostly receive-blocked while waiting for client requests and spends very little time processing them, very few position samples can be obtained and so the results would be unreliable.

To count function calls, your binaries must be compiled and linked with `-p`; for details, see "Enabling call count instrumentation".

NOTE:
You can run an application with profiling instrumentation from the command line. However, using the IDE is more convenient because the Application Profiler automates the requesting of data from the instrumented binary and presents the results in an easy-to-read format.

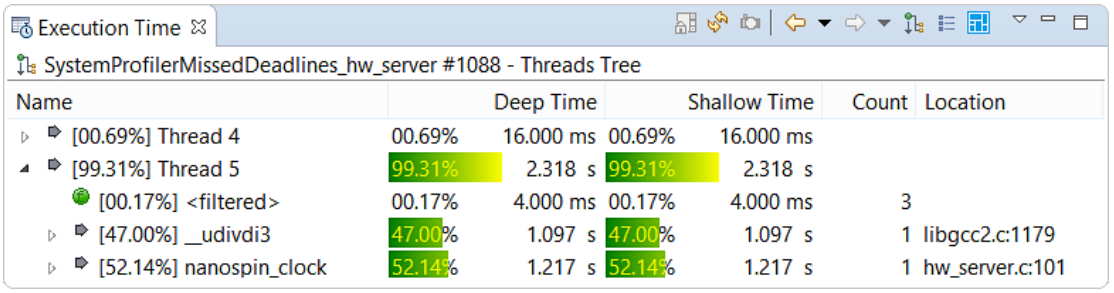
To profile an application by sampling positions and counting calls:

1. In the launch bar, expand the Launch Configuration dropdown (which is in the middle) and select the project that you want to profile.
2. In the Launch Target dropdown (on the right), select the target for running your application.
3. In the Launch Mode dropdown (on the left), select Profile.
4. Click the Edit button () on the right of the Launch Configuration dropdown.
5. In the configuration editor window, click the **Profile** tab on the right and the Application Profiler radio button near the top of this tab.
6. Click the **Sampling** radio button under **Profiling Method**, and verify that **Single Application** is selected under **Profiling Scope**.
7. **Optional:** To see call counts for each function, check the **Use Call Count Instrumentation** box in the **Options** panel.
   Enabling this UI setting makes the IDE check that any application binary being run was built with `-p`. If it wasn't, the IDE doesn't launch the application and instead displays an error.
8. Click **OK** to save the configuration changes and close the window.
9. In the launch bar, click the Profile button ().

The IDE switches to the QNX Analysis perspective. If necessary, the IDE first builds the binary. Then, it uploads the binary and starts running it on the target. At this time, a new session is created and displayed in the **Analysis Sessions** view. As the application runs, profiling results are sent to the IDE, which stores them in the new session and presents them in the **Execution Time** view.

The profiling results include the function runtimes, call counts, and call sites. Initially, function information for all program components is listed. You can expand the **Analysis Sessions** entry and click a component to filter the display. For example, you may want to see only those runtimes for functions in your program code and not in any libraries that it uses, so you can click the application binary.

By default, the **Execution Time** view shows the threads tree, which lists the functions by thread:



You should be able to see statistics for functions in any shared libraries used by your application. If you don't, you must manually define the library paths.

With position sampling, there's no distinction of deep versus shallow time, so both columns report the same metric. In addition to execution times, these columns display numbers and bar graphs representing the percentages of time spent in the various functions. You can change the columns displayed to see nondefault metrics, as explained in the Execution Time reference.

Like the System Information statistics, the profiling results are refreshed every five seconds (by default), so you can closely monitor changes in execution time breakdown as you interact with the application. Double-clicking on any listing with a file and line number shown in the **Location** column opens the source code at that location. The IDE displays a bar graph and the execution time in the left margin.

NOTE:
You can concurrently profile as many applications as you like and multiple instances of the same application. The results for each execution run appear in their own Application Profiler session, independently of other sessions.

Page updated: August 11, 2025

Application Profiler

Application Profiler is a QNX tool that allows you to view the profiling results generated by instrumented binaries that run on a QNX OS target. The results tell you who called each function as well as how often, how long was spent in each function, and how much CPU time individual lines of codes used. This helps you locate inefficient areas in your code without following its execution line by line.

How to configure Application Profiler

When the launch mode is Profile, the Application Profiler is enabled by default. For other launch modes, you can enable the tool manually through the launch configuration **Tools** tab (Run, Debug, or Attach modes) or **Memory** tab (Memory mode). It isn't supported by the Coverage or Check modes.

The tool settings are organized into the following four panels:

- **Profiling Method**
- **Profiling Scope**
- **Options**
- **Control**

Profiling Method

Functions Instrumentation
: Provides precise function runtimes, based on function entry and exit timestamps reported by the binary. The binary must be compiled and linked with the options described in "Enabling function instrumentation".

Sampling
: Provides runtime estimates based on statistical position sampling driven by timer interrupts. Your application doesn't need to be recompiled but must run for a long time for the statistics to be accurate. Also, backtrace information (call stacks) may not be available. To see call counts, the binary must be compiled and linked with the option described in "Enabling call count instrumentation".

Profiling Scope

Single Application
: Allows you to profile a specific process for an extended period of time, but doesn't provide information about context switches.

System Wide
: Writes profiling data in kernel events, which are logged to an output file that you can view in the System Profiler tool. When this option is enabled, the IDE initiates a kernel event trace when launching the application. The trace captures kernel activity from all processes on the target. Any kernel event trace should be limited to a few seconds because longer traces produce too much data to be useful.

  This option is available only when the Profiling Method is **Functions Instrumentation**. When this option is enabled, you must also enable the System Profiler tool by checking its box at the bottom of the launch configuration tab.

Options

These options are available only when the Profiling Method is **Functions Instrumentation** and the Profiling Scope is **Single Application**.

Data transfer method

- **Save on the target, then upload** — Saves all profiling data on the target, then uploads the data file to the host when the program finishes. This setting means you can't see results as they come in, but avoids the overhead of sending data over the network while the program runs.
- **Upload while running** — Sends profiling data to the host while the process runs on the target. This setting lets you see results as they come in, without waiting for the program to finish, but imposes more overhead.

Path on target for profiler trace
: Sets the location of the profiling results file on the target. You can include the `${random}` string, which the IDE replaces with a random number. The IDE can do this replacement in multiple, simultaneous sessions. You can enter either an absolute path that includes the filename or a relative path. If you enter a relative path, the file is written to **/tmp**.

Remove on exit

Removes the profiling results file from the target when the session is complete. If the IDE has substituted a random number for the string `${random}` in the filename, the file is permanently removed.

Use pipe

Creates a pipe file on the target instead of a regular file. A pipe file stores the collected data using less memory and disk space. To use this option, ensure that the `pipe` utility is running on the target. For more information, see `pipe` in the *Utilities Reference*.

The IDE can create the file on the real filesystem only. It can't create it on **/dev/shmem/**.

Profiling Counter

Specifies which counter the instrumentation code uses to determine the current time at function entry or exit points.

- **Time: Default** — Use *ClockCycles()* for single-core and realtime clock for multi-core (default).
- **Time: Clock Cycles** — Use *ClockCycles()* for multi-core, which is faster and has better resolution than the realtime clock.

  NOTE:
  We require that the hardware underlying *ClockCycles()* be synchronized across all processors on a multicore system. This means that threads are no longer required to be bound to the same CPU.

- **Time: Clock Monotonic** — Use the realtime clock, which reports the monotonic count of elapsed time since the system booted.
- **Time: Clock Process** — Use the process time clock.
- **Time: Clock Thread** — Use the thread time clock.
- **Memory: Allocated Heap** — Use allocated heap memory as a profiling counter (enables memory profiling).

Control

These options are available only when the Profiling Method is **Functions Instrumentation**, for either a single application or a system wide profiling scope.

Automatically start profiling

Specifies whether profiling starts automatically or only after you click **Resume Profiling** in the Analysis Sessions view. If you uncheck this box, you must define signals for starting and stopping profiling by checking the box just underneath. Otherwise, no profiling results are captured.

Install start/stop hooks

Enables the **Resume Profiling** and **Pause Profiling** controls in the Analysis Sessions view. If your code already uses the default signal numbers that start and stop profiling, you can specify different values.

- **Pause signal number** — Specifies the signal number that pauses profiling.
- **Resume signal number** — Specifies the signal number that starts or resumes profiling.

NOTE:
Below the four panels, the **Configure Shared Libraries Paths** link takes you to the **Libraries** tab. Use this tab to define the paths of any shared libraries for which you want to see symbol information in the Application Profiler results.

How Application Profiler results are presented

When you launch an application with the Application Profiler enabled, the IDE switches to the QNX Analysis perspective and opens the **Execution Time** view, which shows the application's execution time by function. The IDE also creates a session for storing the profiling results, and displays this new session in the **Analysis Sessions** view.

Analysis Sessions view

This view displays all sessions from analysis tools run on a project within the current workspace. Each session has a header containing the tool icon, binary name, session number, and launch time. Sessions from the Application Profiler, Code Coverage, Memory Analysis, and all Valgrind tools are listed in this view, from the newest (at the top) to the oldest (at the bottom). The session number gets incremented each time a program is run with any of these tools.

For profiling sessions, the icon is a circle with a small clock and checkmark (). Double-clicking the header opens the session (if it's not open) and displays function information from all components in the **Execution Time** view. To filter the results, expand the header and click the appropriate component. For example, to display results for functions in your program code but not in any libraries that it uses, click the application binary.

When you select an active profiling session, which means the application is still running, the view toolbar provides **Resume Profiling** () and **Pause Profiling** () controls that let you start and stop profiling. You can specify signals to act as start and stop hooks for these controls, through the Control panel. The **Take Snapshot** button ( ) is also active, and it allows you to capture the current data without stopping the profiling activity. You can then compare the snapshot data with the final results or other snapshots.

Execution Time view

This view displays the results of profiling sessions, using a table in which each row provides statistics for one function. The table has these default columns:

- **Name** – The name of the function.
- **Deep Time** – The time it took to execute the function and all of its descendants. This metric is also referred to as Total Function Time. It is the pure realtime interval from when the function started until it ended, which includes its shallow time, the sum of its children's deep times, and all time in which the thread wasn't running while blocked inside of it. If the function was called more than once, this column contains the sum of all runtimes when it was called from a particular stack frame or parent function.

  Inside each column entry, on the left, a green bar and percentage value indicate the relative execution time. On the right, another value indicates the absolute execution time.

  For Sampling mode, this column isn't used.
- **Shallow Time** – For Function Instrumentation mode, this column is the deep time minus the sum of its children's runtimes. It roughly represents the time spent in this function only. However, it also includes the time for kernel and instrumented library calls and for profiling the code.

  For Sampling mode, it's an estimated time, calculated by multiplying an interval time by the count of all samples from this function.

  The column entries have a green bar overlaid with a percentage value (on the left) and another numeric value (on the right). These items provide similar information as in the **Deep Time** column.
- **Count** – The number of times the function was called.
- **Location** – The location of the function in the code.

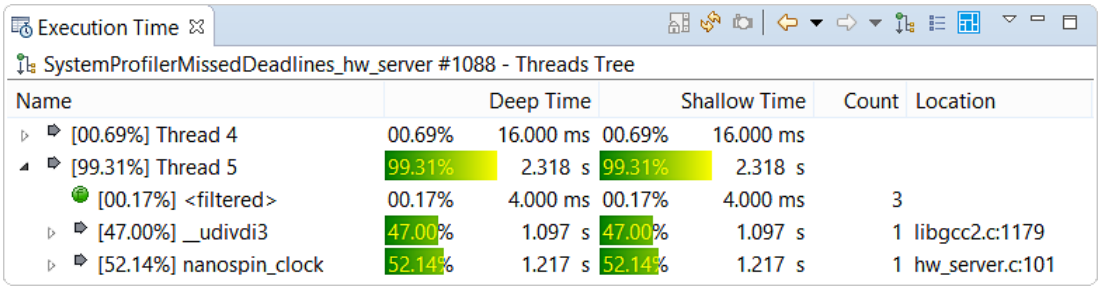
To add or remove columns, click the View Menu dropdown () in the view toolbar, then click **Preferences**. You can add or remove any of the following columns:

- **Percent** – The percentage of Deep Time compared to the Total Time (or compared to the Root Node Time).
- **Average** – The average time spent in the function.
- **Max** – The maximum time spent in the function.
- **Min** – The minimum time spent in the function.
- **Time Stamp** – A timestamp indicating the last time the function was called, if present.
- **Binary** – The binary filename.

NOTE:
You can right-click in any row (i.e., function listing) and select context menu options to display details about the function's call chains; for more information, see "Call sequence information".

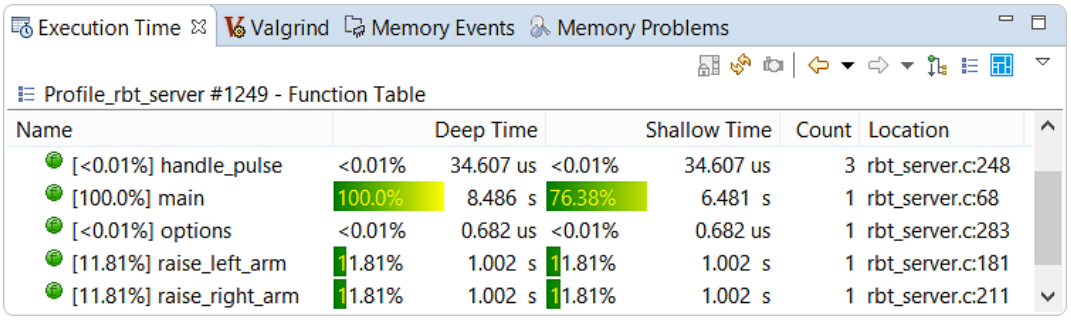
The **Execution Time** toolbar provides actions including but not limited to:

- Disable Automatic Refresh () – Pause the data display in its current state until you unlock it.
- Refresh () – Refresh the data display.
- Take Snapshot and Watch Difference () – Create a profiling session that is a snapshot of the current data. Later, you can compare the results of two sessions to see the effects of any changes you made between execution runs.
- Show Threads Tree () – Display a graphical representation of the threads and calling functions within your application. This display helps you see exactly where the application spends its time and which functions are most used. You can examine the details down to the lowest function calls.



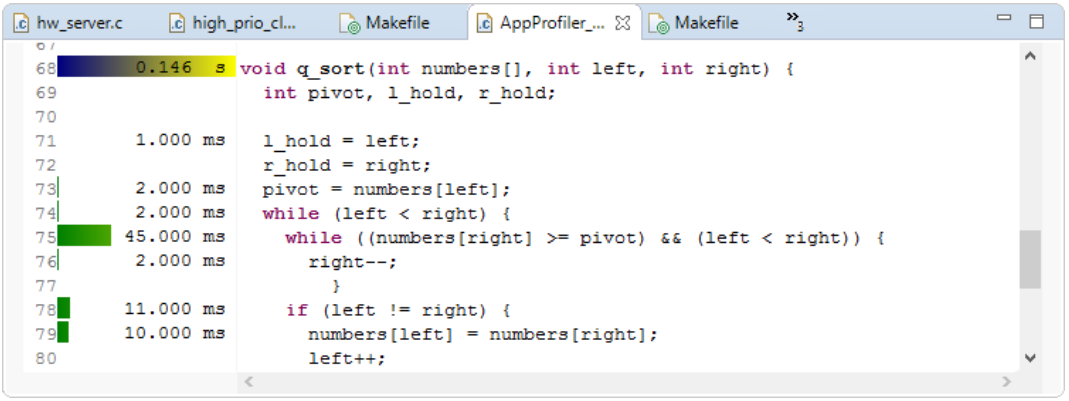
> **NOTE:**
> Information about individual threads is not available for postmortem profiling. Instead, only one thread (with all time assigned to it) is displayed.

- Show Table ( ) — Display a list of functions within your application. This display helps you identify which functions take the longest to complete. In Functions Instrumentation mode, calls to C library functions such as *printf()* are not displayed.



Annotated source

If your executable binary has debug information and your host has the source file for a particular function, you can double-click its entry in the **Execution Time** view to open an annotated version of the code in the editor. The editor shows solid green and graduated blue-yellow bars in the left margin.



> **CAUTION:**
> You may receive incorrect profiling data if you changed the source since the last time you compiled. This is because the annotated editor relies on the line information provided by the debug version of your binary.

The length of the bar represents the percentage of total execution time. The color represents details specific to the function or single line, as follows:

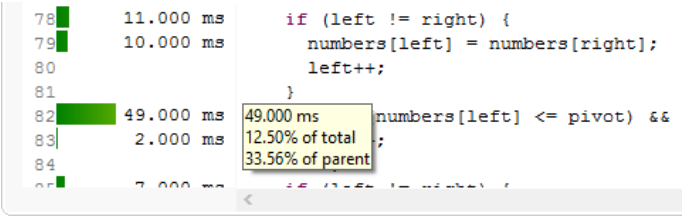
Blue-Yellow

For the first line of a function, a blue-yellow bar shows its total runtime. This includes the function's shallow time, the sum of its children's deep times, and all time in which the thread wasn't running while blocked inside of it.

Green

For a line of code, a green bar shows the amount of time for the inline sampling or call-pair data. The lengths of the green bars within a function add up to the length of the blue-yellow bar on its first line.

To view quantitative profiling data, hover the pointer over a colored bar. The resulting tooltip shows the total number of milliseconds and total percentage of execution time for that code. For children, the tooltip shows the percentage of time relative to the parent.



Call sequence information

You can right-click a table row (i.e., function listing) and then click context menu options to see how the current function fits into the program's call sequence. The options include but aren't limited to:

Show Calls

List all functions called by the selected function. The resulting list lets you examine specific call chains to find which ones have the greatest performance impact. You can expand the entries of descendant functions to see how execution time is distributed among them.

In the called functions list, the columns that are shown by default or that can be selected through the **Preferences** dropdown option are the same as those for the original profiling results. Also, you can right-click again and choose this same option or **Show Reverse Calls** (if it's available), to navigate up or down a particular call stack.

> **NOTE:**
> For this and other options, you can use the **Go Back** (⇦) and **Go Forward** (⇨) toolbar buttons to move between the menu action results and original profiling results.

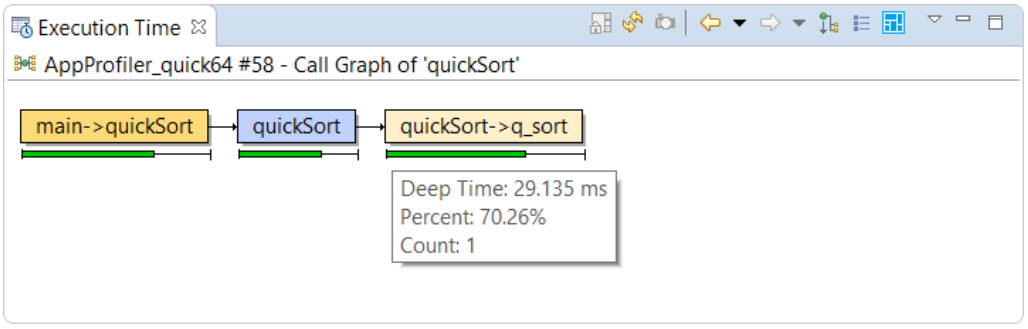
Show Reverse Calls

List all callers of the selected function. The resulting list shows the execution times for all calling functions. The same columns are shown, and you can right-click again and choose this same option or **Show Calls**, to navigate up or down a particular call stack.

You can show the reverse calls only for functions compiled with profiling instrumentation.

Show Call Graph

Display a graph of how the functions are called within the program.



This graph represents functions with colored boxes. The selected function appears in the middle, in blue. On the left, in orange, are all functions that called this function. On the right, in lighter orange, are all functions that the selected function called.

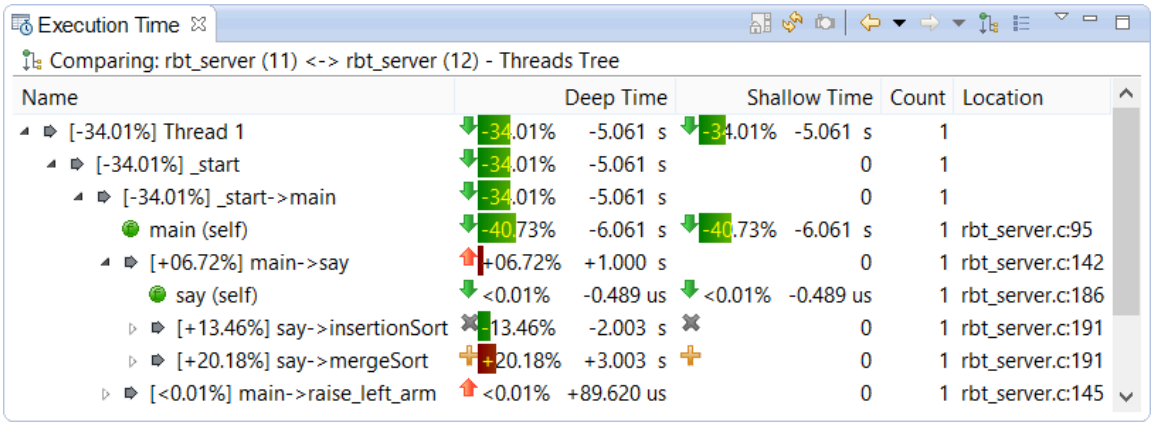
To see the calls to and from a function, click its box in the call graph. If you hover the pointer over this box, **Deep Time**, **Percent**, and **Count** information is displayed, if the information is available.

You can show the call graphs only for functions compiled with profiling instrumentation.

Interpreting differences in session results

You can compare the results of two profiling sessions to see the effect of changes you made to an application. The comparison feature calculates and displays the differences in function runtimes and other metrics between the sessions, in the **Execution Time** view.

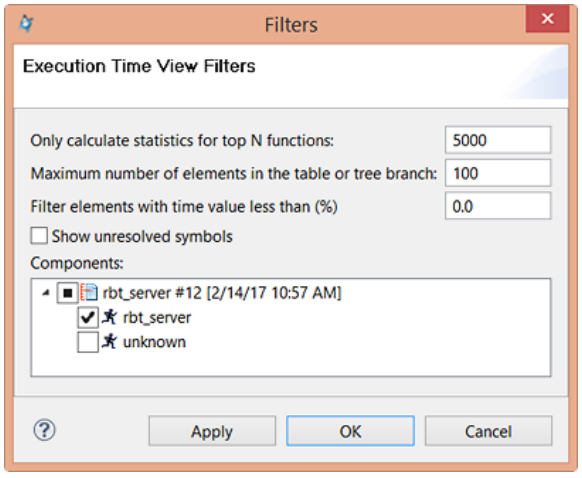
In the comparison results, the columns show the profiling measurement changes from the older to the newer session (i.e., the new values minus the old ones). If a function ran in only one session, its values from that session are displayed.



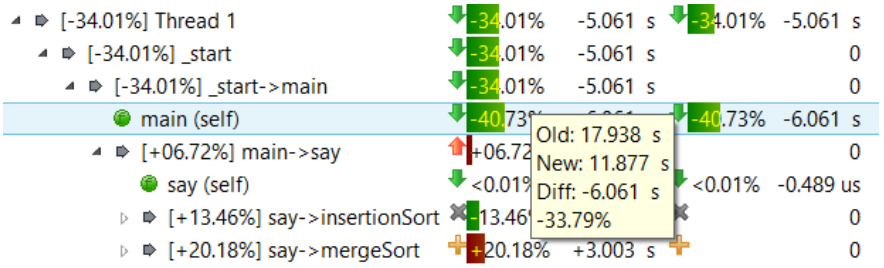
In each column, on the left of the green bar and numeric values that represent the execution time, an icon shows the overall change from the first to the second session. The icon is one of the following:

- – Function runtime has decreased.
- – Function runtime has increased.
- – Function was called in first session only.
- – Function was called in second session only.

To hide insignificant results (e.g., <1% difference) or apply other filters, click the View Menu dropdown (), then click **Filters**. The IDE opens a window with several fields for filtering the profiling results:



You can obtain more information from the tooltips. If you hover the pointer in a **Deep Time** or **Shallow Time** column, the IDE displays a popup message with the old and new time values as well as their absolute and relative differences:



Page updated: August 11, 2025

Adding debug symbols and instrumentation code to binaries

The active build configuration determines which `make` commands are used to compile and link binaries in the current project. To debug and analyze your program with certain tools, you must define the right options in those commands.

When you use the IDE to create a project based on standard makefiles, the IDE generates a default makefile that defines the compiler and linker flags required to build different binary versions. These flags are stored in variables with descriptive names (e.g., `CCFLAGS_debug`, `LDFLAGS_profile`).

If you create a project outside of the IDE or inherit a project from a third party, your makefiles might not be set up in this way. In this case, we recommend reorganizing them to use such variables because it makes it easy to support various analysis activities. You should define variables for storing compiler options, linker options, and libraries to link, for each of four binary versions:

- debug
- release
- coverage
- profile

No tool-enabling options should be used with the release version because it's intended for use in post-deployment (i.e., customer) environments. Also, for some activities, such as profiling, you may have to change the flags and rebuild if you want to measure different metrics.

For QNX Legacy Recursive Make projects, which use managed recursive makefiles, you must adjust the project properties to build the binary for debugging or for generating analysis data. You can't edit the makefiles directly.

Enabling debug symbols

The QNX build tools (`qcc` and `q++`) use the same flag for adding debug symbols as the GDB tools (`gcc` and `g++`), namely, the `-g` option. For standard makefiles, you simply add this option to the debug compiler flags. For example, if you're using the C compiler:

```
CCFLAGS_debug += -g -O0 -fno-builtin
```

Or if you're using the C++ compiler:

```
CXXFLAGS_debug += -g -O0 -fno-builtin
```

For QNX Legacy Recursive Make projects, you must ensure that the debug binary versions are selected for building:

1. Choose **Project** > (and then)**Properties** > (and then)**QNX C/C++ Project**.
2. In the **Build Variants** tab, expand the listings for all architectural variants that you want to build.
3. Check the **debug** boxes for all variants that you want to debug.
4. Click **OK** to confirm the settings and exit the properties window.

   The IDE updates the makefile and rebuilds the project.

Enabling Code Coverage instrumentation

If your project uses standard makefiles, you must define `-f` options to add instrumentation code for measuring which lines of code are executed. You should verify that the following options (shown in bold) are defined in the coverage variables:

```
CCFLAGS_coverage += -g -O0 -ftest-coverage -fprofile-arcs
LDFLAGS_coverage += -ftest-coverage -fprofile-arcs
```

The above example uses `CCFLAGS_coverage`, which is meant for the C compiler, `qcc`. If you're using the C++ compiler, `q++`,you must use `CXXFLAGS_coverage` (and `LDFLAGS_coverage`).

The relevant options are:

- `-O0` (“big-oh zero”) – turns off compiler optimization; optimization eliminates some lines of code, making it impossible to maintain separate execution counts for each line
- `-fprofile-arcs` – adds code to the object files to record program arcs (flow)
- `-ftest-coverage` – generates notes files (**.gcno**), which are needed to report program coverage

For QNX Legacy Recursive Make projects, you must modify the QNX C/C++ Project properties as follows:

1. Choose **Project** > (and then)**Properties** > (and then)**QNX C/C++ Project**.
2. In the **Options** tab, under **Build Options**, check **Build with Code Coverage**.

   You must have only one architecture and variant selected in the **Build Variants** tab. Otherwise, the IDE displays an error and won't build the project until you fix the configuration.
3. In the **Compiler** tab, under **Code Generation Options**, set Optimization Level to “No optimize”.

   Code optimization should be turned off because it can produce inaccurate coverage results. Also, in this release, you don't need to explicitly specify the `-f` options for the compiler flags (`CCFLAGS` or `CXXFLAGS`) or linker flags (`LDFLAGS`).
4. Click **OK** to confirm the settings and exit the properties window.

   The IDE updates the makefile and rebuilds the project.

Full details about all UI fields that control the building of these types of projects are given in the QNX C/C++ Project properties reference.

Enabling call count instrumentation

The `qconn` agent periodically samples the execution position of a running process, by recording the current address every millisecond. This means that you don't need to instrument the binary to see where in the code your application spends most of its time. However, runtimes estimates based on statistical sampling don't tell the full story because you still need to know how often a function gets called when determining what areas to optimize.

In standard makefiles, to insert code that counts function calls, you must add the `-p` option to the debug compiler and linker flags:

```
CCFLAGS_debug += -p -g -O0
LDFLAGS_debug += -p -nopie
```

The above example uses `CCFLAGS_debug`, which is meant for the C compiler, `qcc`. If you're using the C++ compiler, `q++`, you must use `CXXFLAGS_debug` (and `LDFLAGS_debug`).

NOTE:
If a user wants to use call count and **Sampling** profiling in postmortem mode (see postmortem analysis) while using **gmon.out**, the binary should not be Position Independent Executable (PIE). When using `qcc`, it means passing `-nopie`. To insert code that counts function calls, you must add the `-p` option to the debug compiler and linker flags in standard makefiles.

NOTE:
The debug variables, not the profile variables, are used because this type of profiling is done with the debug variant of the binary. With function instrumentation (explained below), the profiling variant of the binary is needed, so different variables must be set in that case.

For the compiler, the `-p` option makes it insert code at every function entry to gather call information. This information allows the IDE to display who called that function and its place in the program's call graph, through the right-click menu of the **Execution Time** view. For the linker, the option makes it link in the profiling version of **libc**. Note that with the default makefile content, you may have to create the `LDFLAGS_debug` variable.

When the **Sampling** profiling method is selected in the Application Profiler UI fields, the IDE always shows runtime estimates in the results, even if `-p` isn't specified. But in this case, neither the call counts nor the call information previously mentioned are shown. Also, if you enable **Use Call Count Instrumentation** and then try to run an application binary built without the `-p` option, the IDE doesn't launch the application and instead displays an error.

NOTE:
If your application uses libraries that weren't compiled with `-p`, the profiling results don't include call counts for their functions. If a lot of execution time is spent inside these functions, this profiling method may not be of much value and you may instead want to use function instrumentation. With this other method, the time spent in a non-instrumented library function is charged to the caller.

For QNX Legacy Recursive Make projects, you must adjust the QNX C/C++ Project properties as follows:

1. Choose **Project** > (and then)**Properties** > (and then)**QNX C/C++ Project**.
2. In the **Options** tab, under **Build Options**, check **Build for Profiling (Call Count Instrumentation)**.

3. Click **OK** to confirm the settings and exit the properties window.

   The IDE updates the makefile and rebuilds the project.

Enabling function instrumentation

In standard makefiles, you must define options to add instrumentation code for measuring function runtimes. You should verify that the following options (shown in bold) are defined in the profile variables:

```
CCFLAGS_profile += -g -O0 -finstrument-functions
LIBS_profile += -lprofilingS
```

The above example uses `CCFLAGS_profile`, which is meant for the C compiler, `qcc`. If you're using the C++ compiler, `q++`, you must use `CXXFLAGS_profile` (and `LDFLAGS_profile`).

NOTE:
The profile variables are used because the profiling variant of the binary must be used to enable function instrumentation.

For the compiler, the `-finstrument-functions` option makes it insert code at every function entry and exit. This code calls a profiling library function that records the current time. When displaying the results, the IDE can then calculate the total time spent inside a given function, based on its entry and exit timestamps.

For the linker, the `-lprofilingS` option shown in this example makes it link against the static shared library **libprofilingS.a**, which implements the time-recording methods needed for profiling. Depending on your system, you may prefer to use the static library file. In this case, you would specify `-lprofiling` to link against **libprofiling.a**.

For QNX Legacy Recursive Make projects, you must adjust the QNX C/C++ Project properties as follows:

1. Choose **Project** > (and then)**Properties** > (and then)**QNX C/C++ Project**.
2. In the **Options** tab, under **Build Options**, check **Build for Profiling (Function Instrumentation)**.
3. Click **OK** to confirm the settings and exit the properties window.

   The IDE updates the makefile and rebuilds the project.

Page updated: August 11, 2025

Build configurations

When you run a project through the launch bar, the IDE builds the project, if necessary, before running the binary on the target. The commands used for compiling and linking are determined by the active build configuration.

You specify the project through the Launch Configuration dropdown, which means you're actually selecting a specific launch configuration for a project. Any launch configuration contains a field for setting the build configuration. This field is found in the **Main** tab of the **Edit Configuration** window, which you can access by clicking the Edit button (⚙) on the right of the dropdown.

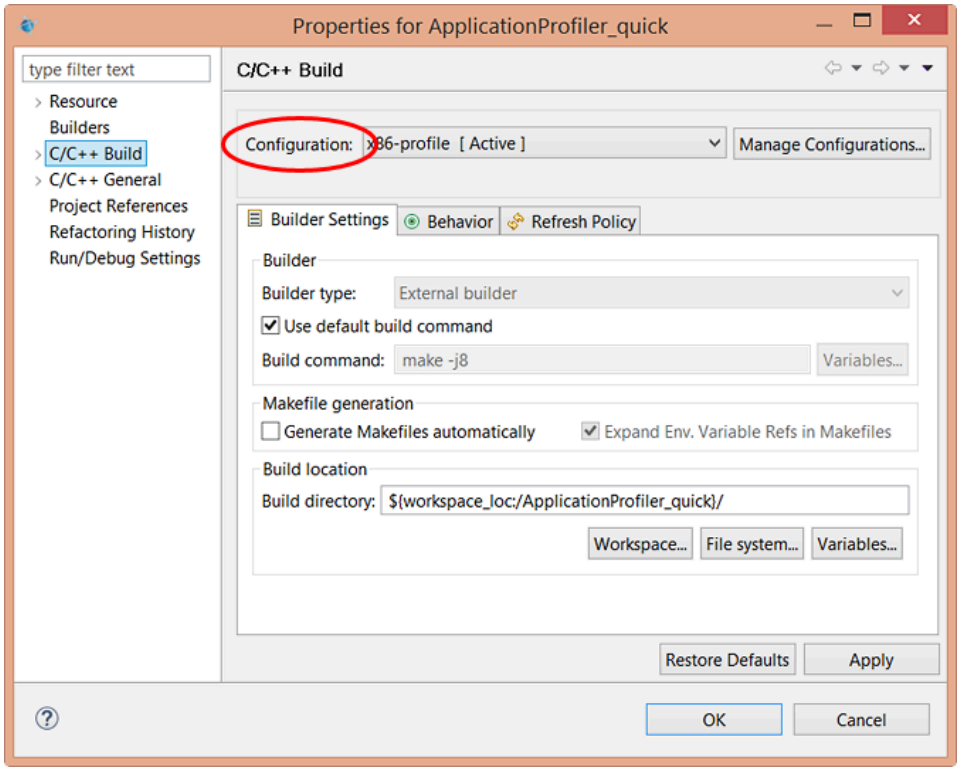
By default, the field is set to **Select Automatically**, which tells the IDE to select the build configuration based on the launch mode. The build configuration determines the architecture variant (e.g., x86_64, AArch64le) and the version (e.g., debug, profile) of the binary that gets built. Different analysis tools require different binary versions, so your selection in the Launch Mode dropdown must support whichever tool you want to use.

The following table shows the binary version built for each launch mode and which analysis tools are supported when building with default launch configuration settings:

Launch mode	Binary version built	Tool(s) supported
Run	debug (contains debug symbols)	GDB debugger (for attaching to the process after it's running)
Debug	debug	GDB debugger (for attaching to the process when launching it)
Coverage	coverage (contains instrumentation code for reporting which lines are executed)	Code Coverage
Memory	debug	Memory Analysis, Application Profiler, or any of the Valgrind tools
Profile	profile (contains instrumentation code for reporting execution positions or function entry and exit times)	Application Profiler
Check	debug	Memory Analysis or any of the Valgrind tools

NOTE:
As the table shows, no launch mode automatically selects the release version of the binary. To build and run this version, you must edit the launch configuration to explicitly select a build configuration based on this version, for whatever architecture variant you want to build. For example, you can set **Build Configuration** to `x86_64-release` to build a stripped binary (i.e., one with no debug symbols or instrumentation code) for a 64-bit x86 target.

In the **Main** tab, you can disable automatic selection by explicitly naming a build configuration, as explained in the **Build Configuration** field description. Next to this field, there's a link with the same name that opens the project properties to the **C/C++ Build** dialog. This dialog lets you control the finer points of building, such as the `make` command and the build directory. You can access this dialog at any time by choosing **Project** > (and then)**Properties** > (and then)**C/C++ Build**. To modify a particular build configuration, you must select it in the dropdown at the top of the main display area:



Page updated: August 11, 2025

Building a QNX project

QNX Tool Suite | Integrated Development Environment User's Guide | Developer | Setup

From the launch bar, you can build a project based on a certain launch mode (i.e., user goal).

NOTE:
The IDE automatically builds the project when you launch it, so you need to perform these steps only if you want to build without launching. You could do this to see if the project builds successfully or to generate a debug version of the binary that's needed by a runtime analysis tool that you intend to run later.

To build a QNX project:

1. In the launch bar, click the button for the Launch Configuration dropdown, which is the middle dropdown:

2. Select the entry for the project that you're building.
   This selection tells the IDE how to build and launch the project. The details of building are controlled by the build configuration.
3. Click the button for the Launch Mode dropdown, which is the leftmost dropdown:

4. Select the launch mode based on your user goal (i.e., whether you intend to later run the application, debug it, or profile it).
   With default project settings, the launch mode determines which version of the binary gets built.
5. Click the Build button ( ) to start the build.

The IDE attempts to build the project and displays the build output in the **Console** window at the bottom. If the build succeeds, you'll see the binary files listed in the **Binaries** folder under the project area in the **Project Explorer**.

The **Problems** window shows any errors (in red) and warnings (in yellow) from the build. You can double-click an error line to open the header or source code file at the line that caused the error.

Page updated: August 11, 2025

Analyzing cache usage with Valgrind Cachegrind

Valgrind Cachegrind simulates cache memory and measures a program's reads and writes and its misses at various cache levels. It also reports branch misprediction rates. Cachegrind writes these statistics into the analysis results, which the IDE then parses to display performance data.

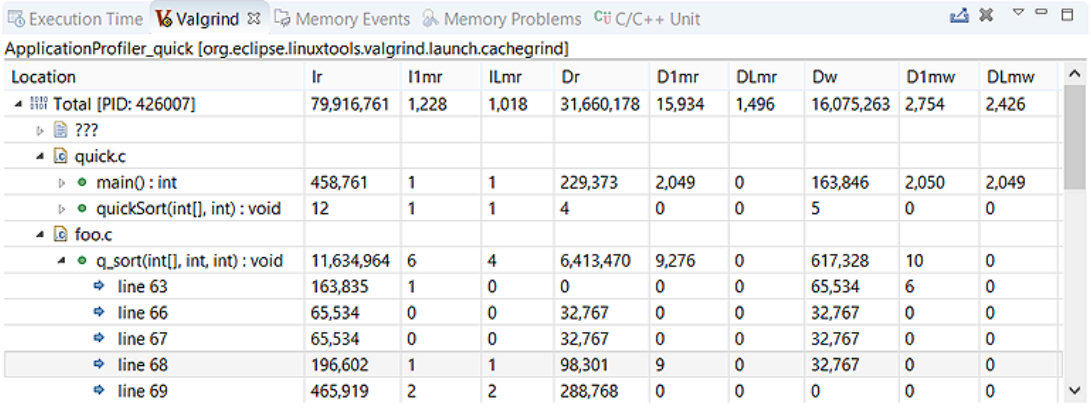
NOTE:
All Valgrind tools can be loaded and run from the command line. However, using the IDE is more convenient because it automates much of the setup by setting Valgrind command options based on UI fields and by copying the analysis results into the host workspace.

To measure cache statistics with Valgrind Cachegrind:

1. In the launch bar, expand the Launch Configuration dropdown (which is in the middle) and select the project for which you want to analyze cache usage.
2. In the Launch Target dropdown (on the right), select the target for running your application.
3. In the Launch Mode dropdown (on the left), select Memory.
4. Click the Edit button (⚙) on the right of the Launch Configuration dropdown.
5. In the configuration editor window, access the Valgrind controls by clicking the **Memory** tab on the right and then the Valgrind radio button near the top of this tab.
6. Select **Cachegrind** from the **Tool to run** dropdown.
7. **Optional:** You can change any settings to customize what gets reported in the Valgrind results.
   The **Cachegrind** tab lets you enable and disable the collection of certain statistics and set the properties of the simulated caches.
8. Click **OK** to save the configuration changes and close the window.
9. In the launch bar, click the Memory button ( ).

The IDE switches to the QNX Analysis perspective. If necessary, the IDE first builds the binary before uploading it to the target. To analyze the application, the IDE instructs Valgrind to execute the uploaded binary with Cachegrind instrumentation. Then, it creates a session for storing the Valgrind results; this new session is displayed in the **Analysis Sessions** view. When the program terminates, Valgrind writes the results to a log file, which the IDE copies into the directory area for the new session.

In the **Valgrind** view, the collected cache statistics are displayed in a table. The top row displays the PID of the Valgrind process in which the analyzed program ran, followed by the total reads, writes, and misses for the first and last cache levels. The rows underneath display per-function statistics. You can click the arrow on the left of a source file's name to see the cache reads, writes, and misses for each function in that file:


Execution Time | Valgrind | Memory Events | Memory Problems | C/C++ Unit

ApplicationProfiler_quick [org.eclipse.linuxtools.valgrind.launch.cachegrind]

Location	Ir	I1mr	ILmr	Dr	D1mr	DLmr	Dw	D1mw	DLmw
Total [PID: 426007]	79,916,761	1,228	1,018	31,660,178	15,934	1,496	16,075,263	2,754	2,426
???									
quick.c									
main() : int	458,761	1	1	229,373	2,049	0	163,846	2,050	2,049
quickSort(int[], int) : void	12	1	1	4	0	0	5	0	0
foo.c									
q_sort(int[], int, int) : void	11,634,964	6	4	6,413,470	9,276	0	617,328	10	0
line 63	163,835	1	0	0	0	0	65,534	6	0
line 66	65,534	0	0	32,767	0	0	32,767	0	0
line 67	65,534	0	0	32,767	0	0	32,767	0	0
line 68	196,602	1	1	98,301	9	0	32,767	0	0
line 69	465,919	2	2	288,768	0	0	0	0	0

Double-clicking any function row opens the source code to the corresponding line.

NOTE:
You can run multiple Valgrind sessions concurrently, using the same tool or different tools, on the same application or different applications. Valgrind log files always contain the PIDs of the Valgrind processes, so their names are always distinct.

Page updated: August 11, 2025

Code Coverage

The Code Coverage tool provides information about which lines of code have been executed. This tool is useful during testing because it tells you whether the code coverage of your test plan is acceptable.

How to configure Code Coverage

You can enable code coverage either by selecting the Coverage mode in the launch bar or by selecting the Run mode, accessing the **Tools** tab in the launch configuration, then clicking the Coverage radio button. These actions make the code coverage fields visible, allowing you to define these settings:

Code Coverage data format

The version of `gcc` code coverage (`gcov`) metrics to collect.

You must select a metrics version based on the SDK you're using to build your program. You can switch between SDKs to build your program using a different SDP installation. For the GCC version supported by a specific SDP release, see the *QNX SDP Release Notes*.

Code Coverage data scan interval

The polling interval, in seconds, used by the IDE to request coverage data from the `qconn` target agent. Frequent polling creates constant network traffic, so you should avoid small values. Usually, the default value of 5 seconds is sufficient.

Comments for this coverage session

Your notes about the session, for your own use. These comments appear at the top of generated reports.

Directory (on target) for temporary coverage data

The target location for writing the code coverage data files. Any binary with code coverage instrumentation outputs a **.gcda** file for each source file in the program. By default, this field names the directory that the executable binary is uploaded to.

The default upload directory is **/tmp** but you can override this setting in the **Upload** tab of the launch configuration. Usually, **/tmp** is linked to **/dev/shmem** so it's writable but temporary, meaning the data files will be gone if the target is rebooted. Note that if the directory is *not* writable, no files are written.

You can click **Browse** to select a target directory from a file selector.

Clean old data on the target (if it exists)

When the box is checked (as by default), the IDE deletes data files that were generated from previous coverage sessions and are stored in the current data directory. If you uncheck this box, the IDE reads these data files when you launch the application and displays the statistics stored in those files in a new coverage session. Then, the displayed statistics are updated to reflect any additional coverage based on the program's latest activity.

Collect data for

This panel lets you select which source files generate code coverage data. The **All sources compiled with code coverage** box is checked by default, which means all source files generate data (assuming they're compiled for code coverage).

If you want to see line coverage data for only some files in the program, uncheck this box and click **Select**. This action opens a popup window that lets you pick the files for which you want to see data.

The **Advanced** button opens the **Remote data collection parameters** window, which lets you select the signal that the IDE sends to `qconn` when requesting coverage data (the default is SIGUSR2). You can choose `Disable dynamic collection` from the dropdown to disable this requesting. There's also a checkbox for suspending the application threads when coverage data is being flushed. By default, this setting is enabled.

How the tool works

When you build a binary with code coverage instrumentation, the compiler inserts code that counts, at run time, each execution of a basic block. Execution of individual lines is *not* counted because the required instrumentation would adversely affect both the size and execution time of the binary. The basic block counts are `gcov` metrics and are sent by the binary to the `qconn` agent.

In this release, the IDE also uses the branch coverage metrics collected by `gcov`. These metrics include the number of times a branch was executed (which is really how many times the branch condition was evaluated), and the number of times each branch was taken. This second metric is quite helpful for ensuring that your tests cover every branch, which is essential for complicated programs.

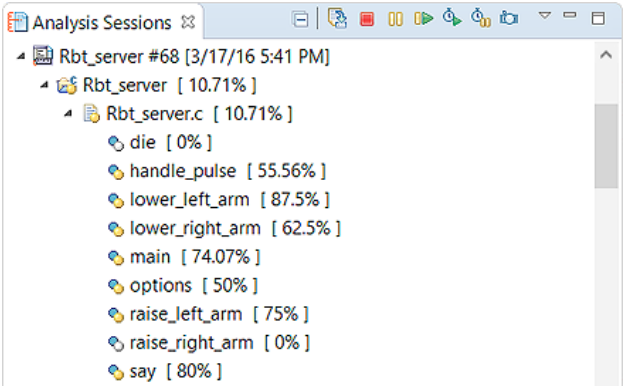
When you launch an application with the Code Coverage tool enabled, the tool requests coverage data from `qconn` at regular intervals, as defined in the launch configuration. The IDE uses the block count data that it receives to calculate the line coverage for the program. To map code blocks to specific lines, the Code Coverage tool needs the notes (**.gcno**) files generated when the binary was built. Then, the IDE visually presents the line coverage data in a new code coverage session.

NOTE:
An instrumented binary always writes the block execution counts to `gcov` data (**.gcda**) files, even if Code Coverage is disabled when you launch the application. Because the tool can read these files, you can import coverage data generated outside of the IDE at any time.

How Code Coverage results are presented

When you launch an application in Coverage mode, the IDE switches to the QNX Analysis perspective and in the **Analysis Sessions** view, displays a new session containing the coverage data for the running program. The session header contains the code coverage session icon (), the binary name, a session number (which gets incremented each time a program is run with any analysis tool that displays its results in this perspective), and finally, a timestamp showing the launch time.

The session contents show the program components, from the application binary to the source files to the functions. Next to each component, the percentage of lines executed is shown in square brackets:



NOTE:
In this IDE version, the latest session is displayed at the top of the view, not at the bottom like in previous versions.

Each code coverage session also has a Logs container that lists an XML-based log file (**session.gcs**), which stores details such as the session name, number, and start time.

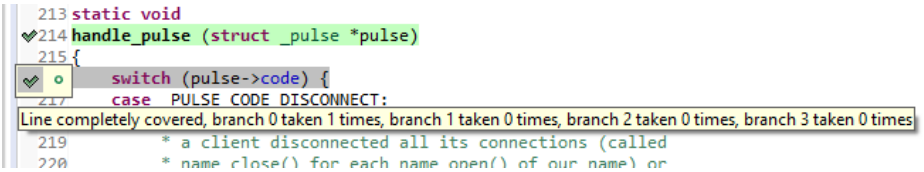
If you double-click a source file or function within a session, the IDE opens the corresponding file in the editor and displays the following markers in the left margin to show line-by-line coverage:

- – the line was fully executed
- – the line was partially executed; if you hover the pointer over one of these markers, the IDE shows what percentage of the line was executed
- – the line was not executed

The right margin displays color bars to indicate line coverage. Because the right margin does not scroll with the rest of the file display, you can use these color bars to quickly locate areas of the file with partial coverage or no coverage (by aligning the scrollbar with the yellow or red bars).

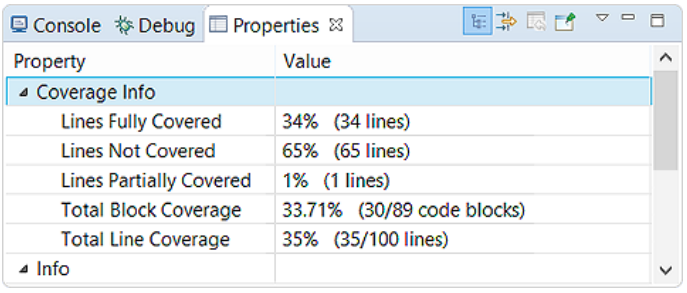
NOTE:
If there's a problem with the coverage data, the IDE might display a warning or error, depending on the code coverage preferences.

Branch coverage is shown when you hover the cursor over a line with branches in the editor. In this release, the IDE shows only branches taken and not branches executed, but this first metric is the important one. For branch numbering, the lowest number is usually the leftmost or topmost branch.



A branch is any programming element that causes the code to not continue executing in a linear way. In addition to the obvious C/C++ branching constructs of `if`, `else`, `for`, `while`, and `switch`, the `&&` and `||` operators introduce their own branches, and there may be others created by calls to inline functions or compiler optimizations.

When you click the application binary in a session, the measurements for the entire program—total block coverage and total line coverage—are listed in the **Properties** view, which is shown in the lower left corner. When you click a file, the view shows the cumulative coverage for all of its functions:



You can view the data from any previous session by expanding it and examining the coverage percentages for its files and functions. If you don't see these components listed when you first expand a session, right-click it and choose **Open**.

How to set report properties

When you select the menu option to export a code coverage session, the IDE displays the **Save Coverage Report** window so you can set properties for the generated report:

Save to directory

The directory the report is written to. The IDE pre-populates this field with the following default path: ***project_dir*/*project_name*__GCC_Code_Coverage_**.

Root filename

The HTML file that provides a summary page for the report. This file is stored in the directory named in the previous field, and contains line coverage measurements for the program, each source file, and each function.

Include source files in report

You can check this box to make the IDE include the source code in the report. In this case, the summary page contains links to the result pages for individual source files and functions.

Include branch coverage in report

You can check this box to make the IDE include branch coverage details in the report.

Color code results

When this box is checked, you can enter threshold values in the text boxes below to visually distinguish between high, medium, and low code coverage in the results. Threshold values let you classify coverage based on the exact percentage of lines executed, instead of the standard metrics of full, partial, and no coverage. Note that the value for high coverage that you enter must be greater than that for medium coverage.

By default, the **Color code results** box is unchecked and so the standard metrics are used.

Open browser on export

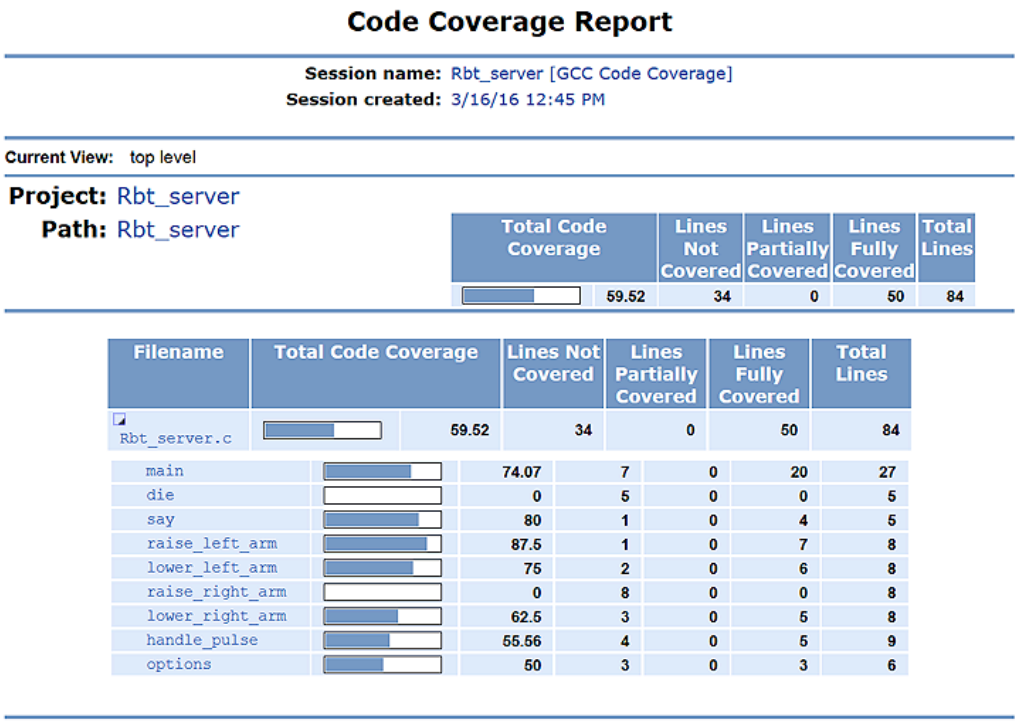
When enabled, the IDE opens the preferred browser and displays the report as soon as the report is generated. You can set the preferred browser through **Window** > (and then)**Preferences** > (and then)**General** > (and then)**Web Browser**, in the **External web browsers** list.

How to read reports

The IDE writes the report as a set of dynamic HTML files. The main output directory stores the HTML root file and the supporting CSS, JavaScript, and icon image files. In a subdirectory named after the project, additional HTML files store the coverage results for source files and functions.

Then, the IDE opens the root file in the default web browser (which you can set in **Window** > (and then)**Preferences** > (and then)**General** > (and then)**Web Browser**). The root file provides a summary of the results.

The table at the top shows measurements for the entire program, including the percentage of total code coverage, the number of lines fully, partially, and not covered, and the total number of lines. The bottom table shows the same measurements for each source file. You can expand a row entry (by clicking the arrowhead next to the filename) to see these measurements for each function in that source file:



Code Coverage Report

Session name: Rbt_server [GCC Code Coverage]
Session created: 3/16/16 12:45 PM

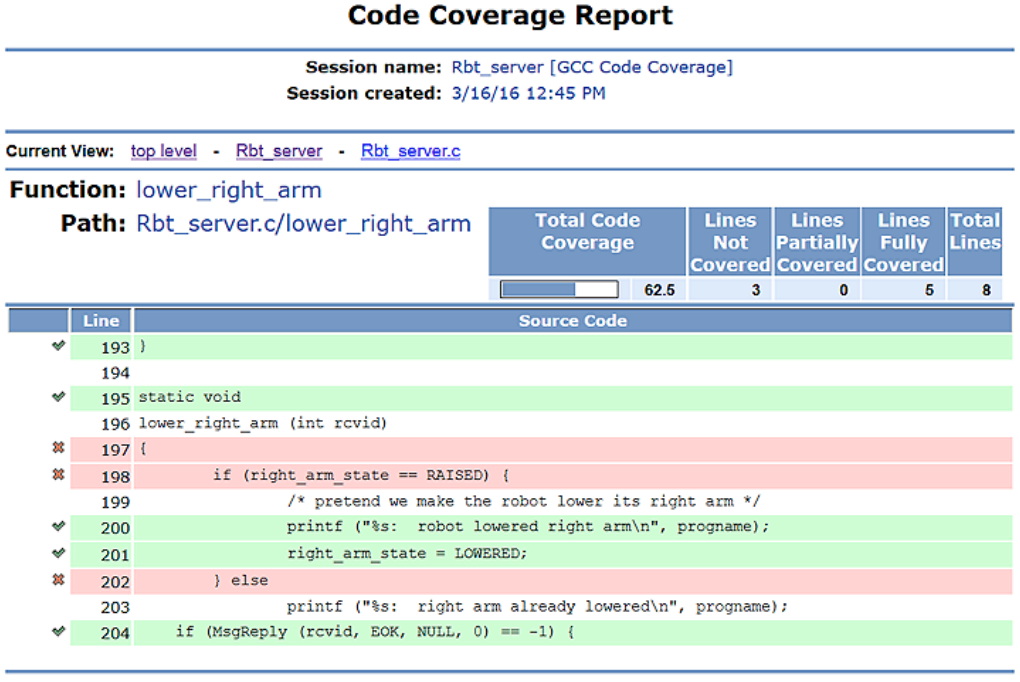
Current View: top level

Project: Rbt_server
Path: Rbt_server

Total Code Coverage	Lines Not Covered	Lines Partially Covered	Lines Fully Covered	Total Lines
59.52	34	0	50	84

Filename	Total Code Coverage	Lines Not Covered	Lines Partially Covered	Lines Fully Covered	Total Lines
Rbt_server.c	59.52	34	0	50	84
main	74.07	7	0	20	27
die	0	5	0	0	5
say	80	1	0	4	5
raise_left_arm	87.5	1	0	7	8
lower_left_arm	75	2	0	6	8
raise_right_arm	0	8	0	0	8
lower_right_arm	62.5	3	0	5	8
handle_pulse	55.56	4	0	5	9
options	50	3	0	3	6

If you enabled the **Include source files in report** setting, the file and function names in the bottom table are also links to the subdirectory files that contain line-by-line coverage information. The report displays this information in the same way as the IDE does in an editor when you double-click a source file or function in a coverage session. You can see coverage markers in the left margin and background coloring that indicates if a line was fully, partially, or not executed:



Code Coverage Report

Session name: Rbt_server [GCC Code Coverage]
Session created: 3/16/16 12:45 PM

Current View: top level - Rbt_server - Rbt_server.c

Function: lower_right_arm
Path: Rbt_server.c/lower_right_arm

Total Code Coverage	Lines Not Covered	Lines Partially Covered	Lines Fully Covered	Total Lines
62.5	3	0	5	8

Line	Source Code
193	}
194	
195	static void
196	lower_right_arm (int rcvid)
197	{
198	if (right_arm_state == RAISED) {
199	/* pretend we make the robot lower its right arm */
200	printf ("%s: robot lowered right arm\n", progname);
201	right_arm_state = LOWERED;
202	} else
203	printf ("%s: right arm already lowered\n", progname);
204	if (MsgReply (rcvid, EOK, NULL, 0) == -1) {

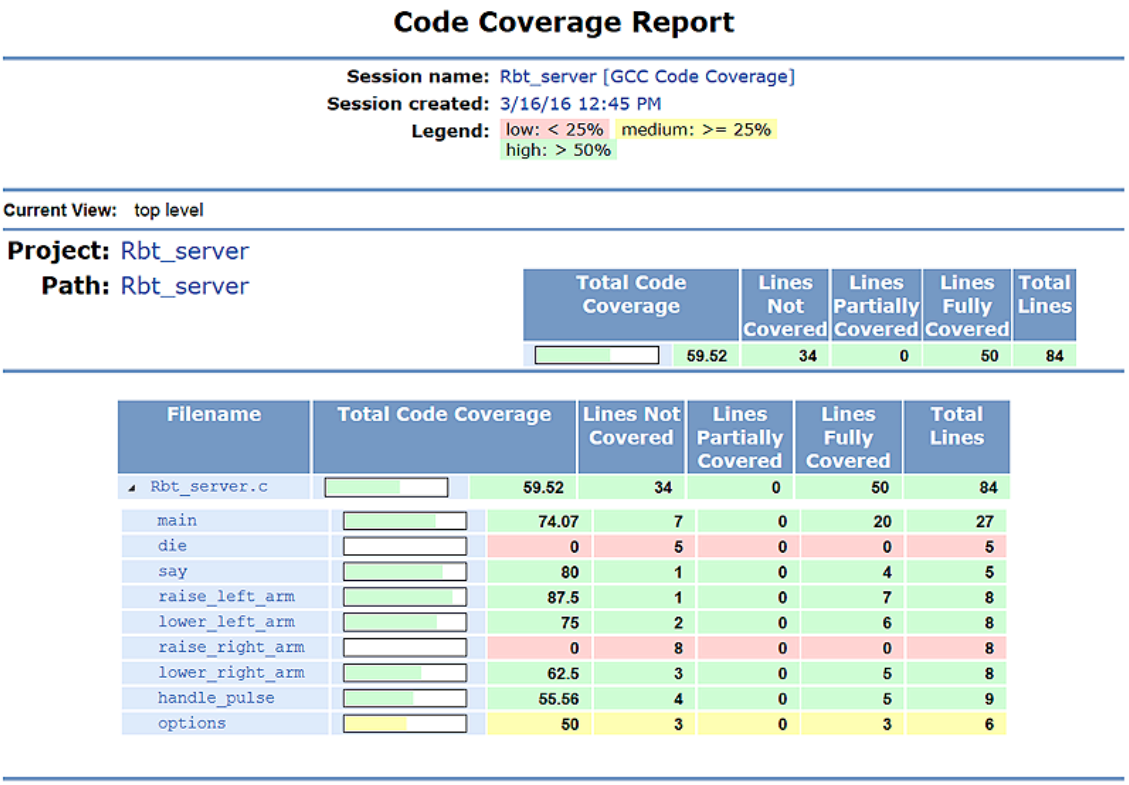
NOTE:

When you open a source file, there's a field, **Source View**, shown just above the code. This field lets you toggle between seeing all of the source code lines or only the uncovered lines. Clicking the latter setting lets you quickly see the areas of code missed by your test program. The **Source View** field isn't shown for functions; when you click a link to a function, you see only its code (with the coverage markup).

You can also add coverage data from included files to the report, by setting the right code coverage preference. When this preference is specified, the summary page has entries for functions in included files (in addition to those functions in the project source files).

If you leave the **Include source files in report** property at its default disabled setting, the IDE still generates HTML files containing the source code and line-by-line markup for files but not functions. You can access these HTML files in the subdirectory named after the project; there just aren't any links to them in the summary page.

If you use default report settings (meaning **Color code results** is unselected), each coverage bar in a table row is colored dark blue and the row cells are colored white. If you enable color coding and define threshold values for different coverage levels, the coverage bar is colored green, yellow, or red based on whether the component had high, medium, or low coverage, and the row cells are colored similarly:



Code Coverage Report

Session name: Rbt_server [GCC Code Coverage]
Session created: 3/16/16 12:45 PM
Legend: low: < 25% medium: >= 25% high: > 50%

Current View: top level

Project: Rbt_server
Path: Rbt_server

Total Code Coverage	Lines Not Covered	Lines Partially Covered	Lines Fully Covered	Total Lines
59.52	34	0	50	84

Filename	Total Code Coverage	Lines Not Covered	Lines Partially Covered	Lines Fully Covered	Total Lines
Rbt_server.c	59.52	34	0	50	84
main	74.07	7	0	20	27
die	0	5	0	0	5
say	80	1	0	4	5
raise_left_arm	87.5	1	0	7	8
lower_left_arm	75	2	0	6	8
raise_right_arm	0	8	0	0	8
lower_right_arm	62.5	3	0	5	8
handle_pulse	55.56	4	0	5	9
options	50	3	0	3	6

NOTE:
The **Color code results** setting affects only the presentation of the table data; the line-by-line display always uses green, yellow, or red markers and line coloring to indicate coverage.

How to set Code Coverage preferences

The **Code Coverage** preference fields are accessed through **Window** > (and then)**Preferences** > (and then)**QNX** > (and then)**Code Coverage**. In the dialog that appears, you can define these Warnings/Errors Display settings:

Ask to show coverage markup for files modified after the session finished

By default, the IDE informs you when the source file you're opening in a coverage session was modified after the session finished. If the file was modified, the coverage markup will likely not match the code in some places because the coverage data is mapped to the line numbers in the code that ran during the session.

If you uncheck this box, the IDE opens any file without advising you if it's changed and hence, the coverage markup might not be accurate.

You can also define these Report Generation settings:

Show code coverage information from included files

By default, code coverage reports don't contain coverage data from included files. You can check this box to add that data to the reports.

Show branch coverage information

This setting determines whether the IDE includes branch coverage information in reports by default. Specifically, it controls whether **Include branch coverage in report** is already checked in the dialog that opens when you select the export option for a coverage session.

Page updated: August 11, 2025

Combining results from multiple sessions

You can combine the results from multiple Code Coverage sessions to view the cumulative coverage across distinct runs of a test program.

To combine the results from multiple sessions:

1. In the **Analysis Sessions** view, use multiselect to indicate the Code Coverage sessions for which you want to combine the results.

   NOTE:
   The sessions that you're combining must be based on the same version of the code. If you changed the code between sessions, the tool can't combine their results.

2. Right-click and choose **Combine**.
3. In the **Combine Coverage Session** window, enter a name for the new results and click **OK**.

The IDE creates a session and displays it in the **Analysis Sessions** view. The data shown reflect the cumulative coverage across all previously selected sessions, meaning the percentages indicate how many lines of code in the given scope were executed by *any* test run. So if a particular line of code were executed in only one of five test runs, this line is still counted as executed.

For lines that were partially executed in some test runs, the combined results show the cumulative coverage for the entire line. Suppose one test run executed 50% of a given line but another test run executed the other 50%. In the combined results, the line is shown as fully (100%) executed. For details on interpreting the results, see "How Code Coverage results are presented".

Page updated: August 11, 2025

Configuring shared library support

The memory-analyzing tools produce results that contain backtraces for allocations, corruption errors, and leaks. To display line numbers for shared libraries in these backtraces, the active tool must have access to library copies with debug symbols.

To find these debug symbols, the tools search the paths listed in the **Libraries** tab. When you add a library to a project, the IDE normally adds the path with the debug version of the library to the search list. It also adds the library file to the list of files to upload to the target, defined in the **Upload** tab. You can strip the debug information when uploading this file—the IDE just needs to find the debug symbols somewhere on the host when displaying the results.

If the automation fails, though, the user must manually specify where to find the right library files. The way to do this depends on the tool.

NOTE:
If you're seeing line numbers for a shared library but double-clicking its function entries takes you to unexpected code locations, you have a mismatch between the host version of the library that contains debug symbols and the target version that gets loaded by the executable. You should therefore rebuild the application to link in the correct version, and verify that the **Upload shared libraries** list (in the **Upload** tab) contains the right library file.

Memory Analysis

If you don't see line numbers for certain shared libraries in the analysis results:

1. Confirm that the project that you want to analyze is selected in the Launch Configuration dropdown, then click the edit button (⚙) on the right.
2. Access the **Libraries** tab and examine the **Shared Libraries Path** listings. For each library, you should see either the directory containing its debug version or the library file itself.
3. If some required directory paths or library files are missing, click the **Auto** button. This action populates the list with the host directories containing any user (non-system) libraries needed by the executable binary.
4. If you still notice missing paths or files, you must manually add them. You can do so with the **Add Directory** and **Add Files** buttons, which open selectors for browsing to and choosing items.
5. When you're finished adding library entries, click **OK** to save the configuration changes and close the window.

Valgrind

By default, the IDE starts a symbol server that the Valgrind tools use to gather debug information. The server runs on the host and searches the paths listed in the **Libraries** tab to provide symbol data to any tool upon request, which the tool uses to write the analysis results.

If Valgrind isn't producing any results and is terminating with a message saying that it's unable to connect to the server, you have to manually start it. To do this:

1. Confirm that the project that you're running Valgrind on is selected in the Launch Configuration dropdown, then click the edit button (⚙) on the right.
2. Access the rightmost tab, confirm that the appropriate Valgrind tool is enabled, then click the **Symbols** tab within the Valgrind controls.
3. Ensure that the **Start symbol server** box is checked and the **Server Port** and **Server Host** fields are properly set.

   You would change the port only if your host were running another service behind the default port of 1500. You would specify an IP address in that second field only if your host had multiple network interfaces and you wanted to use a specific one to communicate with the target.
4. Click **OK** to save the configuration changes and close the window.

You should now be able to analyze a program with a Valgrind tool and see error and memory information about any shared libraries with paths listed in the **Libraries** tab. If the symbol server still doesn't work, you have to upload the shared libraries to the target and specify their paths, as explained below.

Manually loading debug symbols on the target

If the debug symbols are already on the target, you don't need to run the server but you must manually specify the paths containing the debug libraries so Valgrind can load their symbols. To do this:

1. Access the Valgrind controls in the launch configuration.
2. In the **Symbols** tab, uncheck the **Start symbol server** box.
3. In the **Select symbols to be loaded on device** field, add the paths of any libraries used by the application.

   This field is auto-populated with the filenames of any user libraries needed by the executable binary. On the right, there are buttons for adding new library entries, editing the paths in existing entries, removing entries, and moving them in the list. You can uncheck entries, to prevent Valgrind from loading those particular symbols.

Page updated: August 11, 2025

Configuring a QNX target

You can view and change the settings of a QNX target through the **QNX Qconn Target** panel.

You can access this panel through one of the following three methods:

- Selecting, in the Launch Target dropdown, the target connection to configure, then clicking the edit button (⚙) on the right
- Right-clicking the target in the **Project Explorer**, selecting **Properties** from the context menu, then selecting **QNX Qconn Target** in the property window's navigation area
- Right-clicking the target in the **Target Navigator** and selecting **Properties**

The **QNX Qconn Target** panel provides the following UI fields:

Hostname or IP
: The hostname or IP address of the target. You would change this setting only if you reconfigured the network on which the host and target reside.

Port
: The target port behind which `qconn` runs. By default, this is 8000.

Target Attributes
: *(read-only)*

The hardware and operating system specifications of the target.

Qconn Services
: *(read-only)*

The version numbers of the various services within `qconn`.

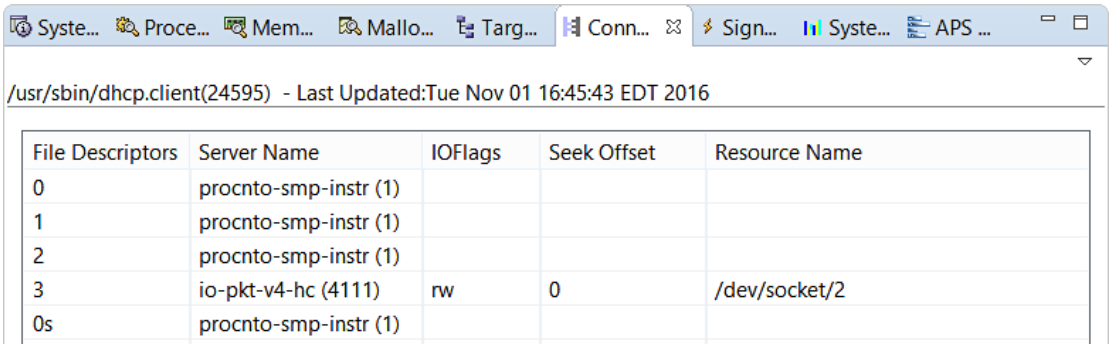
There are also **Restore Defaults** and **Apply** buttons, to reset all fields to their original values and to save the current settings.

Page updated: August 11, 2025

Connection Information

The **Connection Information** view shows the file descriptors, server processes, and IO flags for the connections used by the process selected in the **Target Navigator** view. For some connections, it also shows the pathname of the resource accessed through the connection.

This view lists the file descriptors for all IPC mechanisms, including sockets, named semaphores, message queues, and connection IDs (coids). The information displayed comes from the individual resource manager servers that provide the connections. In some cases, fields are blank because the resource manager doesn't have the ability to return the requested information.



File Descriptors

The file descriptor number. When a file descriptor is created through a side channel, an “s” is displayed beside this number. Because the connection ID is returned from a different space than with standard file descriptors, the ID is greater than any valid standard descriptor.

The connections are sorted by this field, in ascending order. So, the standard input, output, and error connections are listed first, followed by all other connections that were *not* created as side channels, then those that *were* created as such.

To see the full side channel numbers, click the View Menu dropdown (▽) in the view toolbar, then click **Full Side Channels**. For information on side channels, see the *ConnectAttach()* entry in the *C Library Reference*.

Server Name

The resource manager server name.

IOFlags

The read (r) and write (w) status of the file. A double dash (--) indicates no read or write permission. A blank indicates that the information isn't available. For descriptions of additional file status indicators used in this column, see the **fcntl.h** header file.

Seek Offset

The connection's offset from the start of the file.

Resource Name

If available, the pathname of the resource accessed through the connection.

Page updated: August 11, 2025

Controlling a QNX virtual machine

After you create a **QNX Virtual Machine** target connection, you can control the target QNX VM by starting and stopping it, and rebuilding it with the same or different configuration settings.

Starting

To start a stopped VM:

- Right-click the target in the **Project Explorer** or **Target Navigator** and choose **Start Virtual Machine** from the context menu.

  The IDE starts the VM. VMware or VirtualBox VMs start within their respective workstations, and QEMU VMs start in a new console.

Stopping

To stop a running VM:

- Right-click the target in the **Project Explorer** or **Target Navigator** and choose **Stop Virtual Machine** from the context menu.

  If the target is a VMware or VirtualBox VM, the IDE stops and closes the VM. If it's a QEMU VM, the IDE terminates the QEMU process.

Rebuilding

You can rebuild a QNX VM with its current settings. This is handy if you encountered an `mkqnximage` error that prevented the tool from generating the VM, and you've since fixed the error. To rebuild with the current settings:

- Right-click the target in the **Project Explorer** or **Target Navigator** and choose **Rebuild Virtual Machine** from the context menu.

  The IDE calls `mkqnximage` with the same options as before to regenerate the same VM and system image. If this operation succeeds, the IDE restarts the VM.

If you want to change the settings and rebuild the VM:

1. Click the button (⚙) on the right of the target connection name in the Launch Target dropdown.

   This opens the same configuration settings dialog that you used when creating the VM.
2. Edit the settings as needed. For instance, you can enter a different IP address for the IDE to try and connect to.
3. Click **Finish**.

   The IDE calls `mkqnximage` with options based on your newly configured settings to regenerate the VM and system image. If this operation succeeds, the IDE restarts the VM.

Deleting

To delete a QNX VM target:

- Right-click the target in the **Project Explorer** or **Target Navigator** and choose **Delete** from the context menu.

Page updated: August 11, 2025

Copyright and patent notice

QNX Tool Suite | Integrated Development Environment User's Guide | Developer | Setup

BlackBerry Limited
2200 University Avenue E.
Waterloo, Ontario
N2K 0A7
Canada

Voice: +1 519 888-7465
Fax: +1 519 888-6906
Email: info@qnx.com
Web: https://www.qnx.com/

August 11, 2025

Page updated: August 11, 2025

Finding improper memory usage

The IDE includes tools that identify lines of code that corrupt memory or cause thread synchronization problems. This information helps you debug process crashing and improper program results.

When you suspect that memory corruption is making a program misbehave, you should run the debugger and try reproducing the bad behavior. This can help in simple cases, but debugging often fails to find the source of corruption because such errors typically appear in unrelated parts of the program. In this case, you must run tools that track memory allocation and report details about illegal operations.

There's a trade-off between the level of memory checking performed and the overhead imposed by an analysis tool. More memory checking uncovers more problems but slows operations noticeably and may exhaust available system memory. In general, you should try using the tools in the following order:

1. **Valgrind Memcheck**

   This tool checks all memory reads and writes to detect improper memory uses. It reports out-of-bounds accesses and bad pointer parameters but also finds many subtle bugs including:

   - Overlapping memory regions in source and destination pointers
   - Use of uninitialized or freed memory
   - Suspicious parameters, such as negative or excessively large positive values for allocation sizes

   Memcheck doesn't require recompiling the application but does impose substantial overhead, causing the program to run at one-tenth its normal speed (or slower), and increases the memory footprint by two to three times. Also, Valgrind reports the results only when the program terminates, so you can't see corruption errors in real time with Memcheck.

2. **Memory Analysis**

   Based on the data produced by the debug allocation library (**librcheck**), Memory Analysis reports common memory errors, including out-of-bounds data writes and bad pointers given to memory- and string-management functions.

   This tool is easy to set up, not requiring you to recompile the application, and imposes low overhead compared to Memcheck. However, its memory checks are limited, so it may not find many errors.

In multithreaded programs, bad behavior is sometimes caused by a lack of thread synchronization. But tracking down thread synchronization errors is hard because the timing of operations is unpredictable and thus, results aren't always reproducible. You can debug such programs with the following tool:

- **Valgrind Helgrind**

  This tool detects three classes of synchronization errors:

  - Misuses of the POSIX threads (pthreads) API
  - Potential deadlocks caused by incorrect lock ordering
  - Data races resulting from accessing memory without proper locking or synchronization

You must run a binary built with debug information so the active tool can match binary instructions with lines of code and display the symbols (i.e., backtrace) in the results. To see symbols for shared libraries, your host machine must store debug versions of these libraries. Any binary with debug information has a bug icon () next to its name in the Project Explorer. The binary selected for running when you launch a project depends on the launch configuration settings.

If after using the memory-checking tools you still can't find and fix some problems, you should try running a kernel event trace, to find any odd interactions between threads within your program or between your program and another process. Or, you can keep using the conventional debugger to narrow down the cause of improper program results or process crashing.

Page updated: August 11, 2025

Creating a launch configuration

The preferred way of creating a launch configuration is through the launch bar. From here, you can open a wizard that lets you define a new configuration that works with a specific launch mode.

To create a launch configuration:

1. In the launch bar, click the button for the Launch Configuration dropdown, which is the middle dropdown.
2. Click **New Launch Configuration...** to open the **Create Launch Configuration** wizard.
3. In the **Initial Launch Mode** dialog, select the launch mode to use as the default, then click **Next**.
   Although you can change the launch mode through the launch bar after the configuration is created, the selection in this first dialog lets the wizard filter the subsequent list of available launch configuration types.
4. In the **Launch Configuration Type** dialog, select the type of launch configuration to create, then click **Next**.
   Here, each entry accesses a template for defining properties to support building and running a project or to automate a setup or analysis task. For details, see "QNX launch configuration types".
5. Set the desired properties in the **Launch Configuration Properties** dialog.
   For details about the properties applicable to QNX launch configuration types, see "QNX launch configuration properties".
6. Click **Finish** to save the launch configuration and exit the wizard.
   The IDE adds an entry for your new configuration in the Launch Configuration dropdown.

You can select the new configuration whenever you need to perform the underlying task while using the previously specified property settings. You can also edit these settings.

Page updated: August 11, 2025

Creating a non-QNX project

QNX Tool Suite | Integrated Development Environment User's Guide | Developer | Setup

The IDE lets you create any project type supported by the C/C++ Development Toolkit (CDT) in the Eclipse platform.

By default, the non-QNX project types that the CDT supports aren't shown in the common project types listed in the **File** > (and then)**New** submenu, or in the **New Project** window accessible through **File** > (and then)**New** > (and then)**Project**. This design ensures that IDE users see only QNX-specific project wizards when they first use the IDE with a new workspace.

You can display the CDT project wizards by accessing **Window** > (and then)**Preferences** > (and then)**General** > (and then)**Capabilities** and then checking the **CDT Extras** box in the **Activities** panel in the **Capabilities** dialog. For information on using these wizards to create projects, see the “Creating a project” topic in the *C/C++ Development User Guide*.

Page updated: August 11, 2025

Creating a QNX project

QNX Tool Suite | Integrated Development Environment User's Guide | Developer | Setup

Project creation is done through wizards, which set up projects based on user-specified settings.

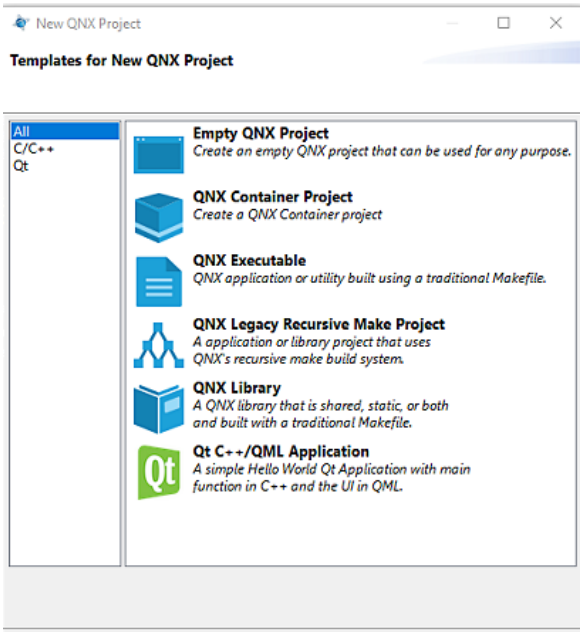
To create a QNX project:

1. Switch to the C/C++ perspective by clicking its button () in the toolbar in the upper right corner.
2. Select **File** > (and then)**New** > (and then)**QNX Project** to launch the QNX Project wizard.
   The **New QNX Project** window appears, with the **All** category selected.
3. Select a project category from the left-side list, click a project type in the display area on the right, then click **Next**.

   The available types are:

Project Type	***Recommended usage***
Empty QNX Project	To create an empty QNX project.
QNX Container Project	To create a QNX Container project.
QNX Executable	To build a QNX application or utility using a traditional Makefile.
QNX Legacy Recursive Make Project	To create an application or library project which uses QNX's recursive make build system.
QNX Library	To create a QNX library built with a traditional Makefile.
Qt C++/QML Application	To create a Qt Application in C++ and QML.

   In this example, we pick the QNX Executable type.



4. In the **New Project** dialog, enter a name in the **Project name** text field.

   NOTE:
   Although the wizard allows it, don't use spaces or any of the following characters in your project name:

   ```
   | ! $ ( " ) & ` : ; \ ' * ? [ ] # ~ = % < > { }
   ```

   These characters cause problems later with compiling and building, because the underlying tools such as `make` and `qcc` don't like them in directory and file names.

   In the same dialog, you can also override the project storage location and specify the working sets that you want the project to belong to.
5. Configure these build settings for the project:
   - the programming language, either C or C++

    - for QNX Library projects, the library type, either shared or static (this field isn't present for our QNX Exectuable example project)
    - the CPU variants, which specify the processor architectures for which the binary files are built
6. Click **Finish**.

The IDE does the following:

- Creates the project (including the necessary source files and makefiles) and makes it visible in the **Project Explorer**.
- Adds an entry for the project in the Launch Configuration dropdown in the launch bar.

At any time, you can right-click the project's entry in the **Project Explorer** to access a context menu that lets you do actions including but not limited to:

- renaming the project
- deleting the project
- adding, deleting, copying, or moving files

For more information, see the **Tasks** > (and then)**Working with projects, folders and files** entry of the *Workbench User Guide*.

Page updated: August 11, 2025

Creating a QNX virtual machine

When you create a **QNX Virtual Machine** target connection, the IDE uses the `mkqnximage` utility to generate a target VM that runs a QNX system image based on settings you specify, such as the CPU architecture and VM platform.

> **CAUTION:**
> If you plan to use VirtualBox as the VM platform and it's your first time using VirtualBox on your host machine, you likely have to create a network interface. For information on doing so, see the `mkqnximage` description in the *Utilities Reference*. If you don't create a network interface, when the IDE finishes creating the VM and tries to start it on the target, the VM will fail to boot and the IDE will output a vague error message.

To create a QNX virtual machine:

1. In the launch bar at the top of the IDE UI, click the button for the Launch Target dropdown, which is the rightmost dropdown.
2. Select **New Launch Target** to open the launch target wizard selector.
3. Select **QNX Virtual Machine Target** in the list of target types, then click **Next** to open the **New QNX Virtual Machine Target** window.
4. In the **Target Name** field, enter a name for your target.
5. Configure any other settings needed for your new VM.
   You can set the following fields:
   - **VM Platform** — the VM platform in which the generated QNX system image will run. The possible platforms are QEMU, VMware, and VirtualBox, but not all VM platforms work with all host OSs. Also, there may be varying levels of support for different VM platforms, in terms of the capabilities of the generated system image. For more information, see the `mkqnximage` reference.
   - **CPU Architecture** — the CPU architecture for which the target VM will be built. To learn how to see which combinations of VM platform and CPU architecture are officially supported for a given host OS, see the `mkqnximage` reference.

     > **CAUTION:**
     > The IDE doesn't enforce the usage of the supported combinations for these settings. You can use unsupported combinations at your own risk because the IDE won't stop you. However, the QNX system image may fail to build or may crash during bootup.

   - **IP Address** — the IP address of the target, which is used by the target connection to talk to `qconn`. If this is left blank, the IDE automatically finds the VM's IP address.
   - **Extra Options** — additional `mkqnximage` options. The `mkqnximage` reference lists and describes the available options.

   Below the configurable fields, there's a read-only text box showing the command line that will be passed to `mkqnximage` based on the options you've set. This field is useful for understanding the relation between your settings and the resulting image, and for extracting the command line if you want to run it manually (i.e., outside of the IDE).
6. Click **Finish** to create the VM.
   The IDE calls the `mkqnximage` utility with options based on your configured settings. The utility tries to generate a VM with a QNX system image, and the operation output is displayed in the **Console** window so you can see any errors and warnings.

   > **NOTE:**
   > If there was a serious error and the VM couldn't be generated, the target connection will still be created but won't be usable until you fix the problem and rebuild the VM. When you restart the IDE, it will try to regenerate the VM.

The IDE adds an entry for the target referred to by the new connection to the Launch Target dropdown, and in the **Project Explorer** and **Target Navigator** views, whose visiblity depend on which IDE perspective is selected.

The IDE also tries to connect to the target using the specified settings. If the connection attempt succeeds, you'll see a purple circle symbolizing a physical connector (even if your target is virtual) but with no red box (), on the left of the launch target name. If it fails, you'll see a red box with an X in the lower left corner of the icon ().

You can now select the new launch target to tell the IDE to debug, profile, or run an application on the QNX VM target. You can also start and stop the VM, rebuild it with the same or different settings, or delete it, as explained in "Controlling a QNX virtual machine".

Page updated: August 11, 2025

Creating a target connection

QNX Tool Suite | Integrated Development Environment User's Guide | Developer | Setup

A target connection defines connection settings for a target machine. Each connection provides a launch target that you can select in the launch bar to specify where an application is to run.

These instructions assume you have a target machine that you plan to connect to through a specified hostname or IP address. There are other target types you can use, as explained in "Supported target types".

For the target connection to be active (and hence, usable for launching applications), you must ensure that the corresponding target is accessible to the host machine through TCP/IP. Information on setting up an IP link with a target is given in "IP communication".

NOTE:
If you are connecting to a target (only tested with VMware target), then you may need to be disconnected from VPN for the IDE to connect to your target.

To create a target connection:

1. In the launch bar at the top of the IDE UI, click the button for the Launch Target dropdown, which is the rightmost dropdown:

2. Select **New Launch Target** to open the launch target wizard selector.
3. Select **QNX Target** in the list of target types, then click **Next** to open the **New QNX Target** window.
4. **Optional:** In the **Target Name** field, you can uncheck the **Same as hostname** box and enter a nondefault name for the launch target.
   For convenience, this box is initially checked so the value entered in the **Connection** field (in the next step) is used as the launch target name.

   NOTE:
   This IDE dialog uses the standard networking term of *hostname* to refer to the name of the machine you're connecting to. In this case, it's the target machine, not the host (development) machine.

5. In the **Connection** field, enter either the hostname or IP address of the target machine.
   The port number is preset and is used by the `qconn` agent, which handles IP communication.
6. Click **Finish** to create the target connection.

The IDE adds an entry for the target referred to by the new connection to the Launch Target dropdown, and in the **Project Explorer** and **Target Navigator** views, whose visiblity depend on which IDE perspective is selected. The IDE also tries to connect to the target using the specified settings. If the connection attempt succeeds, you'll see a purple circle symbolizing a physical connector (even if your target is virtual) but with no red box ( ), on the left of the launch target name. If it fails, you'll see a red box with an X in the lower left corner of the icon ( ).

You can now select the new launch target to tell the IDE to debug, profile, or run an application on the target using the corresponding connection settings.

You can also edit the target connection settings by clicking the button ( ) on the right of its name.

NOTE:
The IDE sometimes displays a read timeout error message when interacting with a target where there are processes that consume a lot of CPU time. This is caused by a timeout of the socket used by the IDE to talk to the `qconn` target agent, which is trying to get data from a busy system. The situations when this occurs include but aren't limited to:

- Sending a signal to a process
- Gathering profiling data from a process
- Terminating a process from the launch bar

Several workarounds can be applied:

- Increase the `qconn` priority (e.g., to 21); its default priority is 10.

- Decrease the priority of the busy process (e.g., to 9); this can be configured in the **Arguments** tab of the launch configuration, when launching the process.
- For a virtual machine target, configure the target to have more cores; this is especially recommended for QNX VM targets.
- Increase the `qconn` socket timeout, by specifying the following line in **qde.ini**: `-Dcom.qnx.qconn.defaultReadTimeout=30000`. This example makes the timeout 30 seconds, but this may not work depending on how badly the process starves the CPU.

Page updated: August 11, 2025

Debugging Applications

QNX Tool Suite | Integrated Development Environment User's Guide | Developer | Setup

The IDE allows you to debug a running application, a library used by the application, or a core file.

Here, *debugging* means finding and fixing problems that prevent a program from completing its tasks or running reliably. This can involve stepping through code but also investigative tasks such as reading memory and CPU statistics to determine which thread is hanging or to understand the conditions around a program crash.

This broader definition means that debugging can be done with tools other than a conventional debugger. In addition to using the GNU Debugger (GDB), you can examine data reported by the System Profiler, check for runtime errors using several memory- and heap-monitoring tools, and see thread states in the System Information.

We first explain the debugging use cases supported by the IDE and how to select the best tool in each case. Then, we describe the workflows that involve using GDB to step through program code in various circumstances. Information about using other runtime analysis tools as part of debugging is given in later chapters.

Page updated: August 11, 2025

Attaching the debugger to a running process

You can attach the debugger to a process running on a target machine.

To debug an active program, you must attach to a binary built with debug information, so the debugger can match binary instructions with lines of code. Any binary with debug information has a bug icon () next to its name in the Project Explorer. The binary selected for running when you launch a project depends on the launch configuration settings.

NOTE:
You can't attach the debugger to a running process over a serial link—your host machine must have an IP connection to your target machine.

NOTE:
All application binaries—executable and library—on your host must match those on your target so GDB can step through the code. To debug, the host and target must run the same SDP build. Even a minor difference in library binaries prevents the debugger from working correctly.

To attach the debugger to a running process:

1. In the launch bar, expand the Launch Configuration dropdown (which is in the middle) and select the project defining the application that you want to debug.
2. In the Launch Target dropdown (on the right), select the target on which your application is running.
3. In the Launch Mode dropdown (on the left), select the Attach mode.
4. Click the Attach button ().
   You don't need to manually enable the debugger in the launch configuration because by default, this tool is enabled when you select the Attach mode. This applies even for new projects on which you have not yet run the debugger.
   The IDE switches to the Debug perspective and opens the **Select Process** window, which lists the processes with the same name as the binary specified in the launch configuration settings.
5. Click the process that you want to attach to, then click **OK**.

The IDE attaches the debugger to the process and stops execution immediately. The **Debug** view displays information about the active threads and their stack frames. More details on this view are found in the **Reference** > (and then)**Debug views** > (and then)**Debug view** entry in the *C/C++ Development User Guide*.

You may not see any source code when the application is initially stopped. For example, if it had just issued a *sleep()* call, there's no code to display. In this case, you can open a source file, add breakpoints, and click the Resume button to make GDB continue executing (after the sleep period finishes) to a certain line of code. You will then see the corresponding stack trace and code displayed.

For information on the debugging controls, see "Debug perspective controls".

Page updated: August 11, 2025

Debugging a child process

By default, when a process creates another process, GDB doesn't follow the execution of the child process. You must attach another instance of the debugger to this new process; this action lets you debug the parent and child concurrently.

NOTE:
If you need to debug only the child and not the parent, you can instruct the `gdb` utility to instead follow the child's code path.

The best strategy for debugging a child process is to make the child send a SIGSTOP signal to itself right after the *fork()* statement that creates it. This stops the process so you can attach the debugger and manually resume execution or step through the child's code.

To debug a child process concurrently with its parent:

1. Set a breakpoint in the parent code path right after the *fork()* statement.
2. In the launch bar, select the appropriate project and launch target, and the Debug launch mode.
3. Click the Debug button ( ).
   The IDE switches to the Debug perspective, which displays the source code being traced and the current variables and breakpoints. If necessary, the IDE builds the application and uploads the binary to the target. Then, it runs the binary and GDB, which it attaches to the binary.
4. If GDB stops on startup, press **F8** or click the Resume button at the top of the **Debug** view to continue execution.
   The program runs until the breakpoint just after the *fork()* call in the parent code path. The child process will now be running and if you added a line to send the SIGSTOP signal, it will be stopped at that line.
5. Learn the ID of the child process.

   In the **Variables** view, you can examine the variable that stores the *fork()* result. Because GDB is following the parent code path, this variable shows the PID of the new child.

   You can also switch to the QNX System Information perspective and in the **System Summary** view, find the processes with same name as the binary you're running. The child process will most likely have a PID that's a slightly higher number than that of the parent.
6. Create a new launch configuration with Attach mode.
   You need to create a new launch configuration to attach a second GDB instance to the child process:
   a. In the launch bar, expand the Launch Configuration dropdown (which is in the middle) and select **New Launch Configuration...**
   b. In the **Launch Mode** panel, select **Attach**.
   c. In the **Launch Configuration Type** panel, select **C/C++ QNX Application**.
   d. Click **Next** to advance to the configuration properties dialog.
   e. Set a name for the launch configuration.
   f. In the **Project** field, set the project that this configuration applies to.
   g. In the **C/C++ Application** field, set the binary file to run.
   h. Click **Apply** and **Finish**. The new launch configuration is selected after you click **Finish**.
7. In the Launch Mode dropdown, select **Attach** using the new launch configuration.
8. Click the **Attach button** ( ).
   The IDE opens the **Select Process** window, which lists the processes with the same name as the binary specified in the launch configuration settings.
9. Click the entry for the child process you want to debug, then click **OK**.
   The IDE runs another GDB instance, attaches it to the child process, and displays an entry for this process in the **Debug** view. If the child's main thread has been stopped by the SIGSTOP signal, the thread label contains the word STOPPED. You can start debugging the child code by pressing **F7** or clicking the Step Return button at the top of the view, once to step out of the *SignalKill()* call and then again to step out of *kill()* and back into the main thread.

You can now debug the child and parent processes concurrently, by alternating between the different sessions in the **Debug** view. When you click a session entry, you access the GDB instance attached to the corresponding process and any debugging actions you then perform (e.g., resuming execution, stepping through code) are applied to that process. This design lets you attach GDB instances to several child processes and debug all of them concurrently.

For information on the debugging controls, see "Debug perspective controls".

Following execution into a child process

When a program executes a *fork()* statement, GDB by default follows the parent code path. To make GDB follow the child code path instead, you must preset the right mode in the `gdb` utility.

In the **Debug** tab of the launch configuration, you can name a command file that GDB executes whenever it starts up. In this file, you can preset the child-following mode using this command:

```
set follow-fork-mode child
```

This strategy is handy when you always want to follow the code path of the child instead of the parent. If you want to debug into the child only occasionally, you can manually set this mode before the child process gets created. To do so, you must first verify that GDB is configured to stop on startup, so you have time to issue the necessary `gdb` command.

To manually follow execution into the child:

1. Select the project, target, and Debug launch mode in the launch bar, then click the Debug button.
   The IDE switches to the Debug perspective, starts GDB, and attaches it to the application process. GDB stops the application on startup, at the first line in *main()* (if you use default settings).
2. Click the **Console** view to interact with the `gdb` utility.
   In this view, the IDE should show the `gdb` output, as indicated by the utility's path given in the view header. If you don't see the `gdb` output initially, click the **Display Selected Console** button ( ) in the upper right area and select the `gdb` binary entry.
3. Enter the command `set follow-fork-mode child`.
   You can confirm this setting by typing `show follow-fork-mode`; `gdb` then displays this message:
   ```
   Debugger response to a program call of fork or vfork is "child".
   ```
4. Press **F8** or click the Resume button in the **Debug** view to continue execution.

GDB lets the application run and create the child process. When this happens, GDB follows execution into the child when the program executes the *fork()* call.

For information on the debugging controls, see "Debug perspective controls".

Page updated: August 11, 2025

Debug perspective controls

QNX Tool Suite | Integrated Development Environment User's Guide | Developer | Setup

The Debug perspective contains many controls for tracing the program's execution and examining its state. Here, we summarize the controls shown in the default views of the perspective and provide the names of the *C/C++ Development User Guide* sections that explain them in full.

NOTE:
The *C/C++ Development User Guide* is part of the Eclipse documentation, and is available in the QNX Momentics IDE help system. The online IDE or QNX SDP documentation doesn't include this programming guide, but you can access it on the Eclipse website at: https://help.eclipse.org/2018-12/topic/org.eclipse.cdt.doc.user/concepts/cdt_o_home.htm?cp=11.

Tracing program execution

(**C/C++ Development User Guide** > (and then)**Reference** > (and then)**Debug views** > (and then)**Debug view**)

The **Debug** view displays the active debugging sessions and for each session, the hierarchy of program components, from the process level to the stack frames of individual threads. Clicking a program component updates the information shown in other views in the Debug perspective.

The buttons along the top support actions that include but aren't limited to:

- Resume ( ) – resume execution of the currently suspended program
- Suspend ( ) – halt execution of the thread currently selected in the program listing
- Terminate ( ) – end the selected debugging session and/or program; the impact of this action depends on what type of component is selected
- Step Into ( ) – execute the current line, including code inside of any routines, and proceed to the next statement
- Step Over ( ) – execute the current line, skipping execution inside of any routines
- Step Return ( ) – continue execution to the end of the current routine, then follow execution to the routine's caller

CAUTION:
It is not recommended to detach the debugger by clicking the Disconnect button ( ). This action sometimes causes the process to abruptly terminate with a nonzero exit code. For example, if you have a *sleep()* call, the process typically aborts when it reaches that line after you detached the debugger.

Accessing variables

(**C/C++ Development User Guide** > (and then)**Tasks** > (and then)**Running and debugging projects** > (and then)**Debugging** > (and then)**Working with variables**)

In the **Variables** view (shown in the upper right area), you can display and change variable values for the selected stack frame.

Managing breakpoints

(**C/C++ Development User Guide** > (and then)**Tasks** > (and then)**Running and debugging projects** > (and then)**Debugging** > (and then)**Using breakpoints, watchpoints, and breakpoint actions**)

You can add breakpoints through the editor, by double-clicking the left margin next to a line of code. When you do so, the new breakpoint appears in the **Breakpoints** view. This view lets you enable or disable breakpoints using checkboxes, and remove and skip them using the right-click menu.

Working with registers

(**C/C++ Development User Guide** > (and then)**Tasks** > (and then)**Running and debugging projects** > (and then)**Debugging** > (and then)**Working with registers**)

The **Registers** view displays the register contents for the selected stack frame, and lets you change their values.

Viewing loaded modules

(**C/C++ Development User Guide** > (and then)**Reference** > (and then)**Debug views** > (and then)**Modules view**)

The **Modules** view displays the executable and libraries loaded for a debugging session.

You can access more debugging controls through nondefault views; for instance, the **Memory** view lets you monitor and change process memory. For the list of debugging views, select **Window** > (and then)**Show View** > (and then)**Other** > (and then)**Debug** in the IDE. For general information on all debugging capabilities of the IDE, see **C/C++ Development User Guide** > (and then)**Tasks** > (and then)**Running and debugging projects** > (and then)**Debugging**.

Page updated: August 11, 2025

Launching an application with the debugger attached

You can launch an application with the debugger attached to step through its entire code path.

To debug applications, your host machine must have an IP or serial connection to your target machine, so the IDE can communicate with the target-side components. Details on setting up these components are given in "Configuring host-target communication".

NOTE:
All application binaries—executable and library—on your host must match those on your target so GDB can step through the code. To debug, the host and target must run the same SDP build. Even a minor difference in library binaries prevents the debugger from working correctly.

To launch an application with the debugger attached:

1. In the launch bar, expand the Launch Configuration dropdown (which is in the middle) and select the project defining the application that you want to debug.
2. In the Launch Target dropdown (on the right), select the target on which you want to debug your application.
3. In the Launch Mode dropdown (on the left), select Debug.
4. Click the Debug button ( ).

The IDE switches to the Debug perspective, which displays the source code being traced and other essential debugging information. If necessary, the IDE builds the application binary (with debugging information) and uploads it to the target. Then, it starts running the binary and GDB, which it attaches to the binary. If configured to do so in the Debug settings, the debugger stops on startup (by default, in the first line of *main()*).

Page updated: August 11, 2025

Debugging libraries

QNX Tool Suite | Integrated Development Environment User's Guide | Developer | Setup

The IDE supports debugging static or shared libraries. You can trace program execution in and out of functions in any library linked to your application.

The only requirement is that you compile the libraries with debug information. If your project uses standard makefiles, you can do so by setting the appropriate flag (command option), which is explained in the `q++, qcc` entry in the *Utilities Reference*. If your project uses recursive makefiles, you must specify a debug build variant through the QNX C/C++ Project properties.

When the project builds successfully, you can start debugging the application and step through code, set breakpoints, and examine variables in library functions as you would in application code. The library debug symbols (i.e., function names and line numbers) should appear in the stack frames of the active threads shown in the **Debug** view. This is because the IDE automatically adds the search paths for any shared libraries to the launch configuration so it can find their debug symbols, which are stored in the same directory as the libraries.

For information on the debugging controls, see "Debug perspective controls".

Page updated: August 11, 2025

Debugging non-IDE code

You can debug code written and compiled outside of the IDE. All you need is a launch configuration—you don't need to create an IDE project to store the code.

The steps shown here work for debugging code written on the same host but built outside of the IDE or even code written (and built) on another machine. For the second case, though, you must have a copy of the source files somewhere on your host, so GDB can display the code during debugging.

NOTE:
For code written in an earlier IDE version, you should import the projects into the new IDE. You can continue developing projects built with recent QNX SDP versions, because the IDE lets you switch between the SDKs of the various SDP installations.

To debug non-IDE code:

1. Create a launch configuration for running the appropriate binary.

   In the **C/C++ Application** field in the **Main** tab, you must provide the full path to the binary located outside of the workspace.

   If you're debugging code that was compiled on another machine, you need to tell the IDE where to find the source files on the host, by defining the appropriate paths in the **Source** tab.
2. Click the Debug button ( ).

The IDE switches to the Debug perspective, which displays the source code being traced and other essential debugging information. Then, it starts running the binary and GDB, which it attaches to the binary. If configured to do so in the Debug settings, the debugger stops on startup (by default, in the first line of *main()*).

For information on the debugging controls, see "Debug perspective controls".

Page updated: August 11, 2025

Debugging a core file

If you're running an application in Run mode (without the debugger attached) and the remote process crashes, the IDE gives you the option of downloading the resulting core file to your host and debugging that file. This is the preferred postmortem debugging method because it's convenient and lets you begin debugging a crashed process immediately.

NOTE:
All application binaries—executable and library—on your host must match those on your target so GDB can step through the code. To debug, the host and target must run the same SDP build. Even a minor difference in library binaries prevents the debugger from working correctly.

Your target must be running the `dumper` utility for a core file to be generated following a crash. The IDE detects the crash based on the program's nonzero exit code and displays a popup window asking whether you want to debug the core file. If you click **OK**, the IDE:

1. Downloads the core file from the target to the host's workspace area, into the project root directory.
2. Creates a C/C++ QNX Remote Core Dump Debugging launch configuration, with a name of "`remote core binary_name`". This new launch configuration, along with the Debug mode and the target on which the process ran, are selected in the launch bar.
3. Switches to the Debug perspective and launches the debugger with the core file attached. The views in this perspective display the active threads and their stack frames as well as the source code at the crash location. This lets you examine the terminated program's final state. For information on the debugging controls, see "Debug perspective controls".

Debugging a locally stored core file

If you obtain a core file from another source (e.g., a customer), you can manually launch the debugger to debug that file:

1. Copy the file to a convenient workspace location, such as the project root directory. If necessary, refresh the **Project Explorer** (by clicking the project and pressing **F5**) to see the copied file.
2. Double-click the core file to open the Core Info editor, which displays the core dump contents, similar to the `coreinfo` utility. This editor also contains a Debug button (⚙) at the top that opens the core file in GDB.

   You can also right-click the file, then select **Debug As** > (and then)**C/C++ QNX Local Core Dump Debugging**.

   NOTE:
   In the earlier IDE versions, you had to right-click the binary executable, choose **Debug As** > (and then)**Debug Configurations**, then provide the path of the core file in a new debug configuration. This release simplifies the loading of core files but still supports that legacy workflow. If you do use this other workflow, be sure to select **C/C++ QNX Core Dump Debugging** as the launch configuration type, not **C/C++ Postmortem Debugger**. This second, Eclipse-based configuration doesn't work for debugging core files produced on QNX-based systems.

At this point, the IDE creates a C/C++ QNX Local Core Dump Debugging launch configuration, with a name of "`core.binary_name`". For this launch configuration type, the IDE uses a special launch target, **Local**, and Debug mode; the launch bar selections are updated appropriately.

Page updated: August 11, 2025

Selecting a tool for debugging

The IDE works with several third-party tools and contains its own features that help you debug improper program results, process hanging, or process crashing.

These debugging use cases are defined as follows:

Improper program results
: No fatal error occurs but the application doesn't do what's expected.

Process hanging
: The application becomes unresponsive but continues running.

Process crashing
: The application exits abruptly, without properly terminating its operations.

Guidelines on selecting a tool for each use case are given below. The debugging capabilities of these tools are listed in "Integrated tools".

Improper program results

There are many IDE tools that can help you find logic errors in program code or memory corruption issues. Generally, you should first launch the application with the debugger attached and try to reproduce the bad behavior. If you can't reproduce it, you can wait until you next see the improper results, then attach the debugger to determine what state the program is in and how it got there. For multiprocess programs, you can debug a child process.

NOTE:
If your program runs too slowly but still does what's expected, you shouldn't run the debugger and look for defects. Instead, you should analyze the program's memory and resource usage and analyze its performance, to determine what changes are needed to make it perform better.

If you step through the program's code and examine its memory and variable contents but can't figure out why the application is misbehaving, you can investigate the issue using other tools:

1. If you think that data is being corrupted, run a memory-checking tool—Memory Analysis, Valgrind Helgrind, or Valgrind Memcheck. These tools identify runtime errors that are often related to memory corruption, such as reads of uninitialized or invalid memory.
2. If you suspect there's a bad value coming from another process or there's an odd interaction between processes, use the System Profiler to see the system-level activity (e.g., kernel calls, interprocess messaging) on the target. To do so, you must first run a kernel event trace to capture this activity. The System Profiler then lets you view the trace data.
3. If both these options fail to identify the cause, you must continue using the debugger. You can set more breakpoints to see which code paths are followed when the bad behavior happens.

Process hanging

When you observe an unresponsive application, the IDE can help you inspect the application's current state. This way, you can learn which processes are hanging without having to rerun the application and attempt to reproduce the problem.

The quickest way to start investigating a hanging process is to view the current target machine state through the QNX System Information perspective. Here, the **System Resources** view lists CPU usage by process, so you can see right away which processes are consuming excessive resources or no resources. You can see thread-level details in the **Process Information** view, including thread states.

If the System Information doesn't give you a good idea of why a process is hanging, you can use other tools to figure out what the application is doing:

1. You can run a kernel event trace to capture the system-level activity on the target and then view the trace data with the System Profiler. These data can tell you if a process is spinning (i.e., actively executing but not making progress) and if another process is involved. An example would be livelock, when the interaction between processes is causing them to cycle endlessly between the same states. The trace data can also reveal deadlock. For instance, if a process is waiting on both a mutex and a semaphore, it's likely a case of priority inversion.

2. To see exactly where the application is stuck, you can attach the debugger to the hanging process. As soon as it attaches to a process, the GDB tool stops execution and shows the current line of code (assuming there are matching binaries on the host). This is particularly helpful for deadlock within a process. Suppose one thread is at the start of a critical section. This likely means it can't acquire a necessary resource and is the cause of the deadlock.
3. Sometimes, attaching the debugger to find the current execution position and stepping through the code doesn't tell you why the program is stuck. In this case, you can use the Application Profiler to enable sampling-based profiling. This profiling method makes the application report its current line at regular intervals, which tells you if a function is consuming a lot of execution time.

NOTE:
The Application Profiler won't help with deadlock because the application must be executing code for the tool to receive samples.

Process crashing

When a process crashes on a machine running `dumper`, a core file is generated and the IDE gives you the option of debugging that core file. If you choose to do so, the IDE downloads it from the target to the host's workspace and launches the debugger. The core file contains the final state of the program, allowing you to see what happened.

If you can reproduce the crash, it's better to launch the application with the debugger attached and do so. The postmortem debugging workflow involving a core file is intended for crashes that are hard to reproduce or for when you don't have access to the device on which the crash occurs.

When the program crashes, GDB stops execution and displays the stack trace, the current line of code, and the last signal received by the program. There are a few options to continue investigating:

1. If you suspect that memory corruption is behind the crash, run a memory-checking tool—Memory Analysis, Valgrind Helgrind, or Valgrind Memcheck. These tools pinpoint problematic lines of code such as illegal memory accesses or double-freeing attempts.
2. If you suspect that a bad value coming from another process or even from a thread within the same process is causing the crash, run a kernel event trace. The System Profiler then displays data that lets you see any odd interactions between threads or processes.
3. If both these options fail to identify the cause, you must continue using the debugger. You can set more breakpoints to see which code paths are followed before the crash occurs.

Page updated: August 11, 2025

Debugging over a serial connection

GDB works over a serial connection, so you can debug a program on a target when a networking link isn't available on that device or when you're debugging networking.

NOTE:
Debugging support is limited for serial connections, because you can start an application process with GDB attached but you can't attach GDB to a running process.

You must configure the target for serial communication before performing the IDE steps to select the target and launch the application with GDB attached.

To debug a program over a serial link:

1. In the Launch Target dropdown, click **New Launch Target** to open the launch target wizard selector.
2. Select **QNX Serial Port Connection** in the list of target types, then click **Next** to open the **New QNX Serial Port Connection** dialog.
3. Fill in the serial port connection settings.
   You can provide any name for your new connection; this name will show up in the Launch Target dropdown. The **Serial port** field must name the host port used to talk over the physical link to the target. The **Baud rate** setting must match the value reported for the serial port on the target; this value is listed when you run `stty` on that target port when configuring serial communication. For the remaining fields, you can adjust their values but the default values shown are the more commonly used settings.
4. Click **Finish** to create the target connection.
   An entry for the new target connection is added to and selected in the Launch Target dropdown. The IDE tries to connect to the target; if it succeeds, the dropdown entry shows a purple circle symbolizing a physical connector but with no red box ( ). If it fails, you'll see a red box with an X in the lower left corner of the icon ( ).
5. In the Launch Configuration dropdown, select the project defining the application that you want to debug.
   In this release, you don't need to create a special launch configuration for serial debugging—you can use the default configuration for any project that supports debugging because this configuration works whether you're debugging over a TCP/IP or a serial link. Also, the Serial Port and Baud Rate properties no longer appear in the launch configuration (because they're set in the target connection).
6. Click the Debug button ( ).

The IDE switches to the Debug perspective, which displays the source code being traced and other essential debugging information. If necessary, the IDE builds the application binary (with debugging information) and uploads it to the target. Then, it starts running the binary and GDB, which it attaches to the binary. If configured to do so in the Debug settings, the debugger stops on startup (by default, in the first line of *main()*).

For information on the debugging controls, see "Debug perspective controls".

When a debugging session ends, the support agent used in serial connections, `pdebug`, always exits. You must then either restart it manually or use the target's reset command (if `pdebug` is launched during startup). The following script shows how to quickly restart `pdebug` to keep it running:

```
while true
  do
     pidin | grep -q pdebug
     if [ $? -ne 0 ]
     then
        echo Start pdebug
        pdebug /dev/ser1,115200
     fi
     sleep 1
  done
```

You can edit the settings for the serial port connection by clicking the button ( ) on the right of its name. In the resulting dialog, you can adjust any of the fields you set when creating the connection.

Page updated: August 11, 2025

Detecting priority inversion

If you notice an application missing its deadlines, it may be experiencing priority inversion. The System Profiler lets you examine thread state change events that happened very rapidly so you can recognize if one thread prevented another thread of a higher priority from running, even for only a short time.

NOTE:
The following section uses the "System Profiling" IDE example. This sample QNX project is accessible through **File** > (and then)**New** > (and then)**Example** > (and then)**QNX Examples** > (and then)**IDE Examples**.

For an explanation of priority inversion, see the "Priority inheritance" section in the *Getting Started with the QNX OS* guide. That topic describes a client-server interaction that emphasizes the need for *priority inheritance*, which entails temporarily setting a thread's priority to the highest priority of any other thread that's blocked (even indirectly) by that first thread. Here, we present a scenario in which a lack of priority inheritance causes priority inversion.

NOTE:
Whenever possible, you should use mutexes instead of semaphores around critical sections. In QNX OS, mutexes propagate priority levels (and hence, support priority inheritance), so they prevent priority inversion. They aren't the same as binary semaphores, which don't propagate priorities. If you inherit code from a third-party or can't reasonably avoid semaphores in your code, you have to follow the process outlined here to detect priority inversion.

Priority inversion can also happen if:

- priority inheritance was disabled for threads created on the channel or for the mutex used
- mutexes are used but the application doesn't assign the right priorities for client and server threads

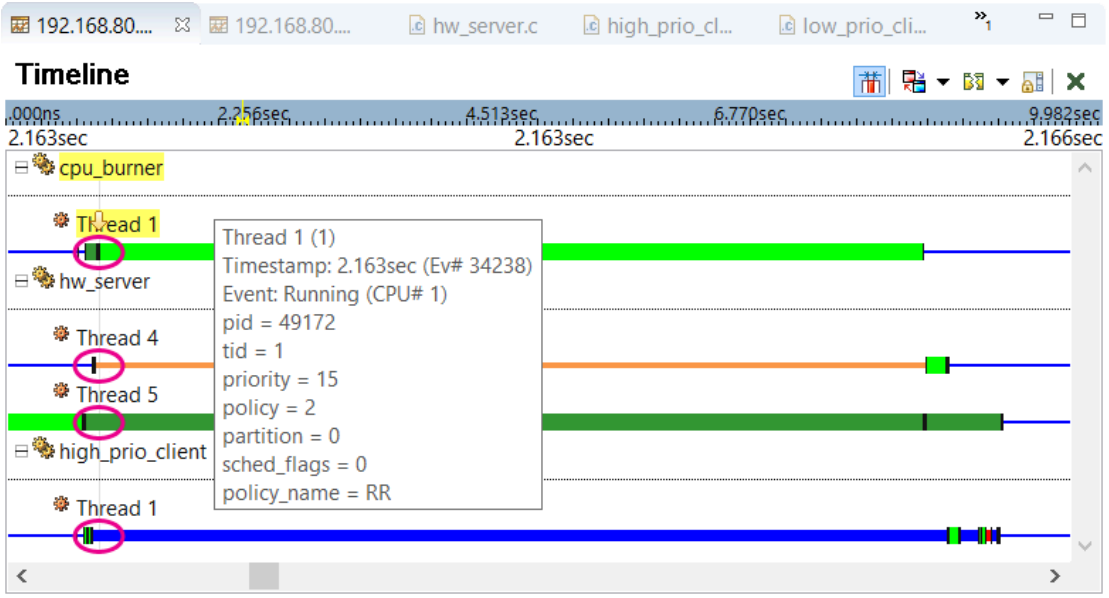
Determining which threads ran at certain times

To detect priority inversion, you must find which higher-priority threads were prevented from running and which lower-priority threads got to run instead. In the Timeline pane, you can examine the thread states at the time your application first became slow or unresponsive.

Initially, the pane displays timelines for all event owners (sources), which include all threads in all processes. This information can be hard to interpret given the large number of owners in most systems. To read the timelines effectively, you must understand your system well enough to know which threads were likely involved in priority inversion. You can then filter the timeline display to show only the relevant timelines.

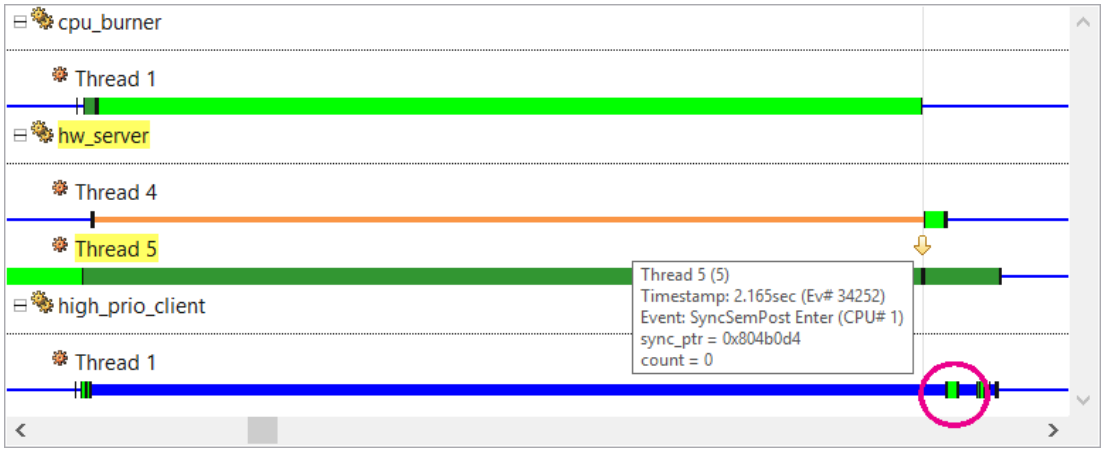
Given that thread scheduling happens very quickly, you must also know fairly precisely when the non-optimal scheduling occurred, likely to within a few milliseconds. To help you find the related event sequence, you can use the zoom feature to examine short timeframes within the trace.

The Timeline screenshot shown below contains the timelines of all `cpu_burner`, `hw_server`, and `high_prio_client` threads from our sample program. Here, the user clicked a Running event in the `cpu_burner Thread 1` timeline and hovered the pointer over the yellow arrow to show the event details:



Based on the state coloring used by the System Profiler, we can see that `hw_server Thread 5` was Ready, `hw_server Thread 4` was Semaphore Blocked, and `high_prio_client Thread 1` was Reply Blocked while `cpu_burner Thread 1` was Running. For readability, you can show the state names by selecting **State Labels** in the Toggle Labels dropdown ( ) in the upper left toolbar. Information about the default mapping of colors to states and how to change that mapping is found in the Timeline State Colors reference. Note that the pink ellipses shown here are callouts in the diagram, to make these color changes stand out, and not actually drawn by the IDE.

From our application design, we know that the server threads each run at level 10 and the client thread runs at level 40. This explains why QNX OS scheduled the `cpu_burner` thread, which runs at level 15, instead of `hw_server Thread 5`. But that particular server thread was blocking two threads, because after it ran later (briefly) and generated events, `hw_server Thread 4` and `high_prio_client Thread 1` were then able to run, as shown in the next screenshot.



Here, the user clicked a SyncSemPost Enter event from `hw_server Thread 5` and hovered the pointer to show the event details. This semaphore post (release) apparently unblocked `Thread 4`, as seen by the color change in this other thread's timeline. This other thread apparently then replied to `high_prio_client Thread 1`, as seen by the client thread's timeline color change. Again, the pink circle here is a callout in the diagram and not drawn by the IDE.

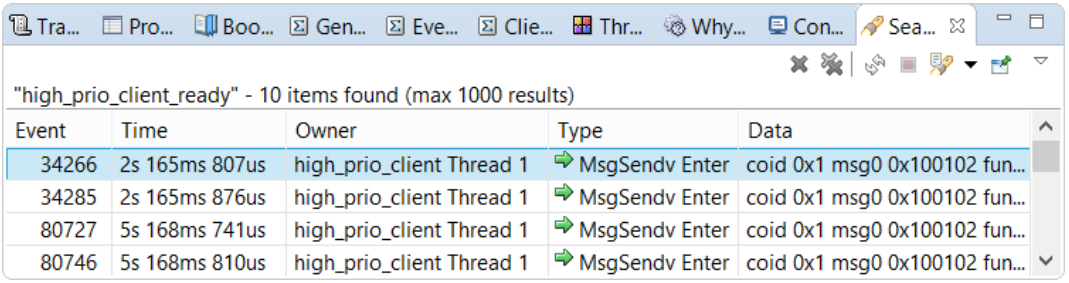
With priority inheritance, the client's higher priority of 40 would have been propagated to `Thread 4` and `Thread 5`. This latter server thread would have run before the `cpu_burner` thread and released the semaphore, allowing the other two threads to complete their work. But because semaphores were used instead of mutexes, no priority inheritance was done and thus, priority inversion occurred.

Finding events that caused priority inversion

The Timeline pane works well if you have a good idea of exactly when priority inversion occurred and which threads were involved. But given the huge number of events in most traces, it can be more practical to search for the events emitted when an application misses its deadlines. When this happens, many applications write messages to a log file or output stream. Our program uses *printf()*, which calls *write()*, which calls *MsgSendV()*. We can then search for the event that's emitted when the kernel enters this last function and that's owned by the high-priority thread that we think was improperly delayed.

Using the **Trace Search** tab of the **Search** dialog (which is accessed through **Search** > (and then)**Search** or **CtrlH**), we create a search condition for MsgSendV Enter events for `high_prio_client Thread 1`. Instructions on creating and configuring search conditions are given in the System Profiler reference.

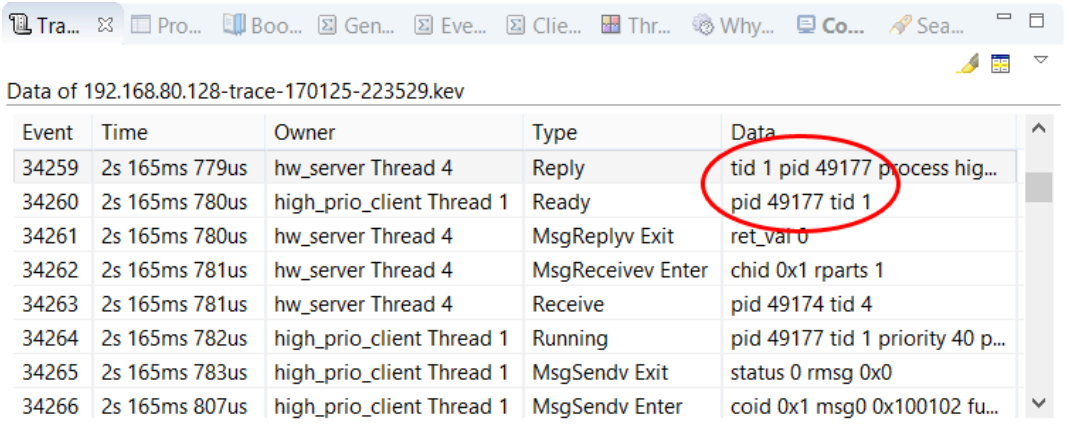
When a search is run, the IDE displays the results in the **Search** view:



Event	Time	Owner	Type	Data
34266	2s 165ms 807us	high_prio_client Thread 1	MsgSendv Enter	coid 0x1 msg0 0x100102 fun...
34285	2s 165ms 876us	high_prio_client Thread 1	MsgSendv Enter	coid 0x1 msg0 0x100102 fun...
80727	5s 168ms 741us	high_prio_client Thread 1	MsgSendv Enter	coid 0x1 msg0 0x100102 fun...
80746	5s 168ms 810us	high_prio_client Thread 1	MsgSendv Enter	coid 0x1 msg0 0x100102 fun...

You can double-click a row in the results to navigate to that event in the timeline display. In addition to highlighting the event with a yellow arrow and dotted vertical line, the Timeline pane draws red circles around it and other events of the same type. The toolbar in the upper left corner of the IDE provides buttons for navigating between events in the timeline.

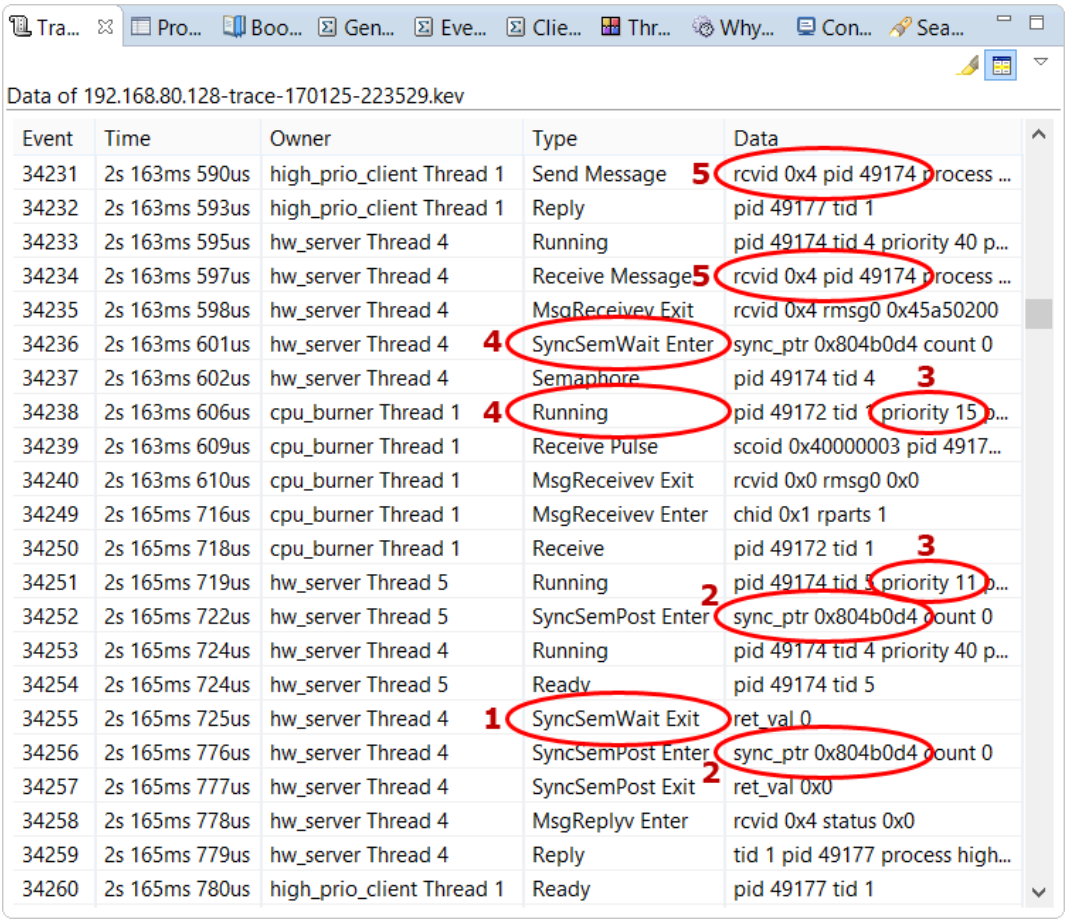
To see the full list of events that occurred, access the **Trace Event Log** view. If the event highlighted in the timeline display isn't selected in this other view, click the **Enable / disable event filtering** button ( ) in the upper right view controls. This filters the event log list so you see only those owners currently displayed in the Timeline pane.



Event	Time	Owner	Type	Data
34259	2s 165ms 779us	hw_server Thread 4	Reply	tid 1 pid 49177 process hig...
34260	2s 165ms 780us	high_prio_client Thread 1	Ready	pid 49177 tid 1
34261	2s 165ms 780us	hw_server Thread 4	MsgReplyv Exit	ret_val 0
34262	2s 165ms 781us	hw_server Thread 4	MsgReceivev Enter	chid 0x1 rparts 1
34263	2s 165ms 781us	hw_server Thread 4	Receive	pid 49174 tid 4
34264	2s 165ms 782us	high_prio_client Thread 1	Running	pid 49177 tid 1 priority 40 p...
34265	2s 165ms 783us	high_prio_client Thread 1	MsgSendv Exit	status 0 rmsg 0x0
34266	2s 165ms 807us	high_prio_client Thread 1	MsgSendv Enter	coid 0x1 msg0 0x100102 fu...

In our case, we want to know why `high_prio_client Thread 1` didn't run sooner. To find out, we click the **Previous Event By Owner** button ( ) to navigate to this thread's last Ready event. If we then click the **Previous Event** button ( ) to navigate to the immediately preceeding event, we see that `hw_server Thread 4` replied to the client thread. This is evident from the matching TID and PID values given in the Reply and Ready events.

The next question is: why didn't the server reply sooner? To answer this, we must navigate backwards through many events:



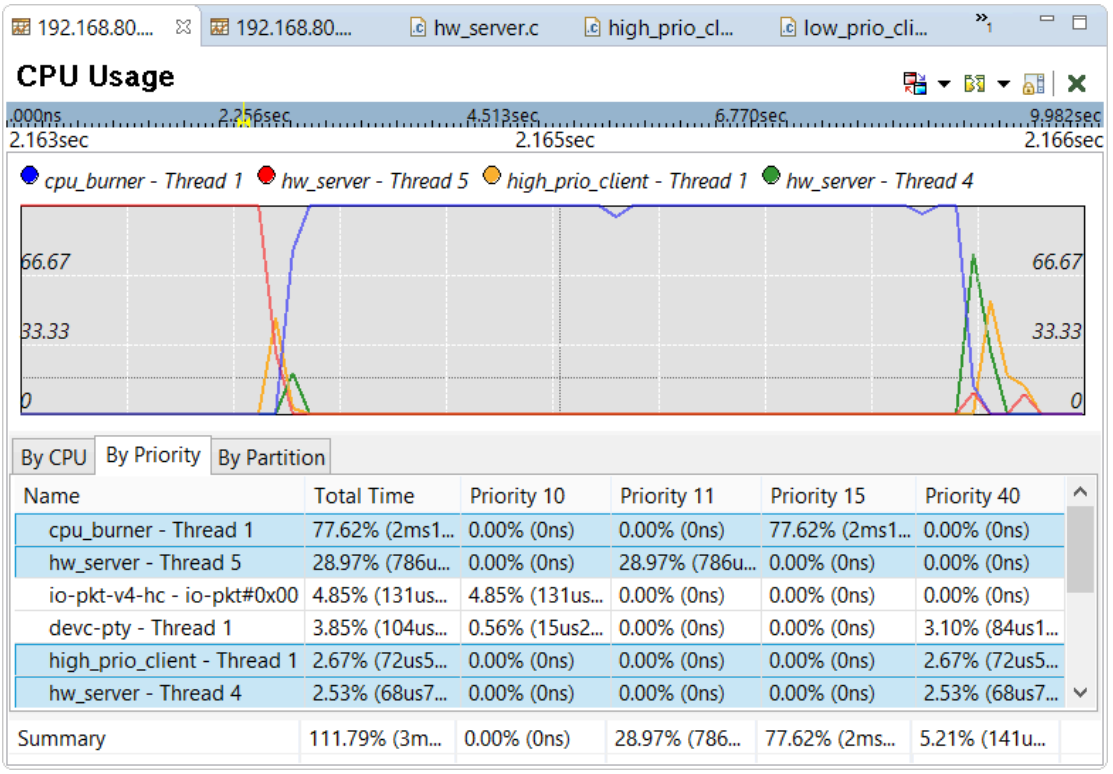
The latest screenshot reveals the following retroactive event sequence:

1. `hw_server Thread 4` was waiting on a semaphore, as indicated by its SyncSemWait Exit event.
2. `hw_server Thread 4` couldn't continue its work until `hw_server Thread 5` released the same semaphore, as shown by the matching addresses in the SyncSemPost Enter events of these server threads.
3. `hw_server Thread 5` couldn't run until `cpu_burner Thread 1` finished running, as seen by the Running events that show the server thread had a lower priority than this other thread.
4. `hw_server Thread 4` was waiting on the semaphore before `cpu_burner Thread 1` was scheduled to run, as indicated by the order of the SyncSemWait Enter and Running events for these threads.
5. `high_prio_client Thread 1` was waiting for a reply from `hw_server Thread 4`, as shown by the matching PIDs in the Send Message and Receive Message events for these threads.

Viewing the CPU usage of threads of different priorities

You can confirm the shift in CPU assignment from the high-priority to medium-priority thread by switching to the CPU Usage pane. To do this, select the option with this name in the Switch Pane dropdown in the editor controls (▾). This pane contains a graph that illustrates the CPU usage history of any thread selected in the table underneath. The time period displayed matches the Timeline selection, so the graph should be zoomed in enough to easily see the CPU usage changes that resulted from the priority inversion.

The following screenshot shows that CPU consumption for `hw_server Thread 5` suddenly goes from 100% to 0%, exactly when the consumption for `high_prio_client Thread 1` and `hw_server Thread 4` also goes to 0%. For a short but noticeable timeframe, none of these threads runs while `cpu_burner Thread 1` consumes all of the CPU:



The **By Priority** tab displays how long the threads ran at specific priorities. Priority inversion is evident because the `cpu_burner` thread ran at level 15 for about 2 ms but `high_prio_client Thread 1` and `hw_server Thread 4` ran at level 40 for well under 1 ms each. Also, the graph shows CPU usage rising and dropping quickly for these two threads at around 2.164 s into the trace. This pattern suggests that the client and server were communicating briefly just before both stopped executing for awhile. Indeed, both threads began executing again after the medium-priority thread finished running at around 2.166 s, presumably because the server was able to do the requested work and reply to the client.

Page updated: August 11, 2025

Developing Projects with the IDE

QNX Tool Suite | Integrated Development Environment User's Guide | Developer | Setup

To use the QNX Momentics IDE to write applications targeted for QNX OS systems, you must define *projects*, which are containers that store your source code, configuration files, and binary files.

The IDE contains wizards for creating projects based on programming language and whether you're writing an application or library. In this guide, we focus mostly on QNX projects, which support C or C++ programs on multiple processors, with standard or recursive makefiles. You can create non-QNX projects that are supported by the Eclipse platform; see "Creating a non-QNX project" for details.

The general procedure of developing and running an application is as follows:

1. Create an application project using the appropriate wizard.
2. Write the code to implement whatever functionality your application is meant to provide. Or, import existing code that provides this functionality.
3. If you want to add libraries to your project, update the makefile as needed.
4. Build the application to produce a binary. Fix any errors and rebuild the application as necessary.
5. Run the application on the target and verify the proper behavior.

These steps are explained in the sections that follow, which demonstrate how to develop and run a "Hello World" application. We don't provide an example of developing a library project, but you can follow steps 1 through 4 to do this.

Page updated: August 11, 2025

Editing a launch configuration

Through the launch bar, you can open the launch configuration manager, which allows you to edit a launch configuration's settings and start the task indicated by the current launch mode.

NOTE:
This release contains the legacy controls in the **Run** menu, **Project Explorer**, and toolbar along the top of the IDE UI for accessing the filtered lists of launch configurations for the Run, Debug, and Profile use cases. However, the new preferred way of editing configurations is to go through the launch bar.

To edit a launch configuration through the launch bar:

1. In the Launch Configuration dropdown, select the configuration you want to edit.
2. Click the Edit button (⚙) on the right.

   The **Edit Configuration** window opens. This window shows all properties of the selected configuration in a set of tabs; the exact tabs and fields displayed depend on the configuration type.

You can click **OK** to save the latest settings and close the window. Clicking **Cancel** reverts to the settings in effect when the configuration was last saved (and closes the window).

Page updated: August 11, 2025

Editors

Editors control how you interact with project files that are opened in the editor pane.

You can open a file by:

- Selecting **File** > (and then)**Open File**, then browsing to and selecting the appropriate file in the file selector.
- Right-clicking the file's entry in the **Project Explorer** (or any of the navigation views) and selecting **Open** from the context menu.
- Double-clicking the file's entry in any navigation view.

NOTE:
If you open a file using this first method, the IDE uses the default editor. If you use either of these last two methods, the IDE uses the last editor selected for the file.

The default editor depends on the file type. For example, header files (**.h**) and source code files (**.c** and **.cpp**) are opened with the C/C++ editor, makefiles with the Makefile editor, and binary files with the Default Binary File editor. This last editor displays the assembly code with the corresponding program statements in comments.

The C/C++ editor facilitates coding in C and C++, providing features such as syntax highlighting, auto-completion, and auto-indenting. It also provides tooltip (hover-based) help; when you position the pointer over the name of a function or macro defined in the standard library, a popup window shows the component's definition.

To open a file using a particular editor:

- Right-click the file's entry in the **Project Explorer**, select **Open With**, then select the desired editor.

  The list of available editors varies with the file type but there are always entries of:

  System Editor
  : Use an external program to open the file. Clicking this entry brings up a list of available system programs, similar to Windows Explorer when you select **Open With** on a file.

  Default Editor
  : Use the default editor for this file type. This option is handy if you previously selected another editor but want to revert to the default editor.

  Other
  : Clicking this entry brings up the **Editor Selection** window, which lists all editors built into the IDE. Note that most file types can be opened only with certain editors.

The file is then displayed in the editor pane. You can open the same file with multiple editors but each editor will display the file in a separate window.

For general information about editors, see the **Concepts** > (and then)**Editors** entry in the *Workbench User Guide*.

Page updated: August 11, 2025

Exporting Application Profiler data

You can export Application Profiler sessions to XML files if you want to share profiling results. You can also export function table data to CSV files if you want to view the data in a spreadsheet application.

The XML files generated by the export feature are self-contained, meaning you can later import those profiling sessions into another IDE workspace or installation without having to name the filepaths of the application binary and shared libraries.

To export Application Profiler sesssion data:

1. Select **File** > (and then)**Export** > (and then)**QNX** > (and then)**Application Profiler Session**, then click **Next**.
   Alternatively, if the **Analysis Sessions** view is open, you can right-click any Application Profiler session and choose **Export Session**.
2. In the **Export** window, select the sessions from which you want to export data.
   The analysis sessions for all tools are listed, from oldest (at the top) to newest (at the bottom), with checkboxes for selecting individual sessions. There are also **Select All** and **Deselect All** buttons.
3. In the **Options** panel, choose the output format.
   By default, the **Export to xml** radio button is selected, which means all session data are written to a single XML file. If you click **Export function table to csv**, the fields underneath become active. You can check **Generate header row** to include column headers that display field names in the output file. The dropdown on the right lets you choose the time unit for reporting function runtimes.
4. In the **Output File** field, specify the file for storing the data.
   You can enter an absolute host path or click **Browse** to choose a directory from a file selector. You must enter a file name in the selector window. The file extension is preset depending on which radio button is currently selected in **Options**.
5. Click **Finish**.
   The IDE writes the data from the selected sessions to the output file, in the format indicated by the **Options** panel. If you specified an existing file, the IDE asks if you want to overwrite it.

For XML files, the session data stored in them can be imported into a different IDE installation, even one of another version. For CSV files, you can view the function data in a spreadsheet application such as Excel.

Page updated: August 11, 2025

Exporting Code Coverage results

Sometimes, it's useful to view coverage data outside of the IDE. The Code Coverage export feature lets you generate a report that contains the data of a coverage session and that can be viewed in a browser.

In this release, there is no longer an action for saving session data to an XML file. Instead, you can export the results from a session into HTML files.

To export Code Coverage results:

1. In the **Analysis Sessions** view, right-click the session for which you want to save the results.
   The session can still be in progress—in this case, the IDE just outputs the results gathered so far.
2. Select **Export Session** from the context menu.
3. In the **Save Coverage Report** window, specify the path for saving the report as well as the name of the root file.

   For convenience, the IDE fills in a default path for saving the report. This path is a subdirectory within the project directory and is named ***project_name*__GCC_Code_Coverage_**. The root file (by default, **index.html**) contains the report summary.

   You can set other report generation properties, as explained in "How to set report properties".

The IDE exports the Code Coverage results into a new report and opens the root file in the default web browser. The report lets you see coverage measurements from the entire program to each source file and function, as explained in "How to read reports".

Page updated: August 11, 2025

Exporting Memory Analysis data

You can export Memory Analysis sessions to XML files if you want to share analysis results. You can also export event data to CSV files if you want to view the data in a spreadsheet application.

Although the Memory Analysis tool produces a log (**.rmat**) file for every session, this file type doesn't have as much information as the XML files generated by the export feature. You can import a log into the IDE, but for the event data to include line numbers (and hence, be readable), your host needs to have copies of the application binary and any shared libraries used during the session. Conversely, the XML file format is self-contained and doesn't require any supporting files on the host.

To export Memory Analysis sesssion data:

1. Select **File** > (and then)**Export** > (and then)**QNX** > (and then)**Memory Analysis Data**, then click **Next**.
   Alternatively, if the **Analysis Sessions** view is open, you can right-click any Memory Analysis session and choose **Export Session**.
2. In the **Export** window, select the sessions for which you want to export data.
   The Memory Analysis sessions are listed from oldest (at the top) to newest (at the bottom), with checkboxes for selecting individual sessions. There are also **Select All** and **Deselect All** buttons.
3. In the **Output File** field, specify the file for storing the data.
   You can enter an absolute host path or click **Browse** to choose a directory from a file selector. You must enter a file name in the selector window. The file extension is preset depending on which radio button is currently selected in **Options**.
4. In the **Options** panel, choose the output details.

   The **Export all** option is selected by default and lets you export all selected sessions to a single XML file. The other four options let you export the data from a specific event type to a CSV file:

   - **Export memory events** — Allocation and deallocation events
   - **Export runtime errors** — Memory errors or leaks detected by the tool
   - **Export bin events** — Events related to bins, which group allocations based on byte size ranges
   - **Export band events** — Events related to bands, which are small blocks of preallocated memory

   If you choose to export the data for an event type, you can check **Generate header row** to include column headers that display field names in the output file.
5. Click **Finish**.
   The IDE writes the data from the selected sessions to the output file, in the format indicated by the **Options** panel. If you specified an existing file, the IDE asks if you want to overwrite it.

For XML files, the session data stored in them can be imported into a different IDE installation, even one of another version. For CSV files, you can view the event data in a spreadsheet application such as Excel. The Memory Analysis reference explains the CSV file contents for exported event data.

Page updated: August 11, 2025

Importing and exporting kernel event trace results

The System Profiler can import kernel event trace results generated outside of the IDE. When you examine trace results in the tool's editor, you can export a subset of events, to share or view the most relevant data. You can also export profiling data generated by instrumented applications into a new profiling session.

Importing kernel event trace results

Unlike the tools that analyze individual applications, the System Profiler doesn't have an import wizard. There are no tool-specific steps for importing trace results. You can simply choose **File** > (and then)**Open File** to open the standard file selector, then choose the appropriate kernel event log (**.kev**) file.

If you start kernel event tracing outside of the IDE, whether through `tracelogger` or an application that calls *TraceEvent()* to manage event logging and data capturing, the resulting log file gets written to whatever target location was specified by the tool. Often, the file is named **tracebuffer.kev** and is found in either **/dev/shmem/** or the directory in which the tool ran. You must then upload the file to a suitable host location using the Target File System Navigator.

You don't need to import the results from a kernel event trace started from the IDE, because the log file is uploaded to the active workspace as soon as the trace finishes. Specifically, the file is copied to the directory for the target connection used to run the trace (i.e., ***workspace_dir/target_conn_name***).

Exporting kernel event trace results

There are no tool-specific steps for exporting kernel event trace results. With the trace results open in the System Profiler editor, you can simply choose **File** > (and then)**Save As** (the editor must be in focus) to save a copy of the kernel event data.

A new log (**.kev**) file is created, based on the currently open log file. You can restrict the data exported to this new file so you keep only the information most relevant for your system analysis. In any editor pane except Summary, you can select a region of the graph (by clicking and dragging the mouse), then export only those kernel events that occurred in that selected timeframe and not the entire trace period. This is done by clicking the **Selected** radio button under the **Event Range** panel in the **Save As** dialog.

You can also restrict the exported data to the event owners and types named in the **Filters** view and currently displayed in the editor pane. This is done by checking the **Apply current filters** box under the **Event Filters** panel in the **Save As** dialog.

The new log file contains the same attribute information as the original log file, including the system version, boot time, number of CPUs, and so on. All events in the new file have timestamps that are relative to the start time of the exported trace data, not the start time of the original trace. For example, suppose that in the original trace, a given process was created exactly 5 seconds since the start. If you then export the trace data from exactly 3 seconds onwards, the new file contains a creation event for this process with a timestamp of 2 seconds.

Regardless of how you ran the kernel event trace, when the log file has been copied into the workspace, you can open it from the **Project Explorer**. The log file is found either in the project for the target connection used to initiate the trace or wherever you uploaded it to. You really need to export the results only if you want to make a copy of them or save a data subset.

Exporting profiling data from trace results

The IDE lets you start an application with function instrumentation enabled and a kernel event trace at the same time. This workflow makes the application log timestamped function entries and exits as kernel events, so its function runtime data appear in the kernel event trace results. When viewing these results, you can export these data into a new Application Profiler session. To do this:

1. With the trace results open in the System Profiler editor, switch to the Timeline pane by clicking the Switch Pane dropdown in the editor controls () and selecting **Timeline**.
2. **Optional:** If you want to export profiling data for only a specific timeframe, click and drag the mouse over the period of interest within the Timeline display. You may have to zoom in or out to see the appropriate timeframe. If you don't select a particular area, event data throughout the trace period are exported.
3. Right-click and choose **Open with QNX Application Profiler**.

   The **Import Application Profiler Data** dialog is opened, and it contains the name of the kernel event log file in the **Import From File** field.

4. In **Executable File**, specify the executable binary for which you want to export profiling data.

   You can manually enter a filepath or click **Browse** to pick a file from the workspace or filesystem through a file selector. The *${workspace_loc:*}* variable is also supported.

5. In the **Libraries Search Path** panel, add the paths of any libraries that the profiled application uses.

   These paths tell the IDE where to find shared libraries referenced by the profiling data. This UI control behaves the same way as the **Libraries** tab in the launch configuration.

   When you've defined the necessary paths, if the profiled application uses source code compiled on another machine, click **Next**; otherwise skip to Step 7.

6. **Optional:** In the **Binary Path** window, specify any additional required source code paths.

   The **Source Lookup Path** panel lists the host paths that the IDE should search to find source code that was part of the instrumented binary but wasn't compiled on the host. This UI control behaves like the **Source** tab in the launch configuration.

7. Click **Finish**.

   The IDE switches to the QNX Analysis perspective and begins importing the data into a new Application Profiler session, which it displays in the **Analysis Sessions** view. The session follows the standard naming convention consisting of the binary that produced the profiling results followed by a new, unique session number. You can rename the session by right-clicking its entry and choosing **Rename**.

NOTE:
You can run as many instrumented binaries as you like while the kernel event trace proceeds. Each of these binaries logs its own Function Enter and Function Exit events to the trace, and you can filter the Timeline display to show the events from any subset of the instrumented binaries. However, when exporting function data, you have to export the data generated by each application into its own profiling session.

Page updated: August 11, 2025

Exporting Valgrind logs

QNX Tool Suite | Integrated Development Environment User's Guide | Developer | Setup

When you run a Valgrind tool on an application through the IDE, the log files generated by the tool are copied to the current workspace when the application terminates. You can then export these log files if you want to share the analysis results or later import them into another workspace or IDE version.

To export Valgrind logs:

1. Select **File** > (and then)**Export** > (and then)**Other** > (and then)**Valgrind Log Files**, then click **Next**.
   Alternatively, if the **Analysis Sessions** view is open, you can right-click any Valgrind session and choose **Export Session**. Note that the session must be open.
2. In the **Export** window, select the log files to export.
   The log files generated during the session are listed with checkboxes next to each file so you can select or deselect it. There are also **Select All** and **Deselect All** buttons.
3. Under **Destination Directory**, specify the path for storing the log files.
   You can enter an absolute host path or click **Browse** to choose a directory from a file selector.
4. Click **Finish**.
   The IDE writes the selected log files to the specified directory.

You can then share the log files with other team members so that they may import them into their own workspaces. Or, you can import them into another workspace or IDE version on your host.

The file contents depend on the tool. For Memcheck, the text output is in a flat file format, which each line listing the target process PID followed by one Memcheck message. For Massif, the output contains the command options given to the tool followed by multi-line sections listing the data from the heap snapshots. More information about the output of these tools is given in the Valgrind User Manual.

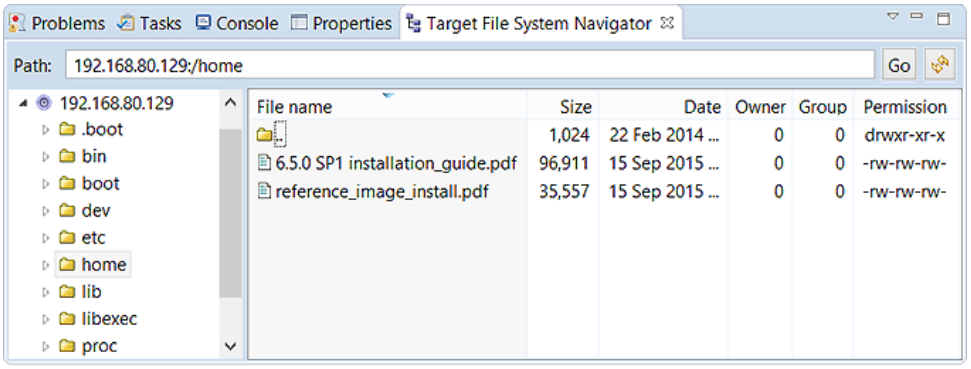
Page updated: August 11, 2025

Target File System Navigator

The **Target File System Navigator** lets you easily copy files between your host and a filesystem residing on a target machine.

This view is shown in the bottom set of views in the default C/C++ perspective. You can also access this view through: **Window** > (and then)**Show View** > (and then)**Other** > (and then)**QNX Targets** > (and then)**Target File System Navigator**

The view shows the target name and directory tree in the left pane, and the contents of the selected directory in the right pane:



Each existing target connection is listed in the left pane, so you must create a connection before you can navigate any target filesystems.

NOTE:
If the **Target File System Navigator** view shows only one pane, click the dropdown button (▽) in the upper right corner and select **Show table**. You can also customize the view by selecting **Table Parameters** or **Show files in tree**. This last option is implicitly selected when **Show table** is *not* selected, so you can see the files in the left pane when the right pane is absent.

Copying files from the host to a target

NOTE:
Before copying text files, you must ensure they have UNIX newline separators (`\n`) because some utilities on the target may not work with Windows newline separators (`\r\n`). The IDE has a handy feature for doing so, which you can access by selecting **File** > (and then)**Convert Line Delimiters To** > (and then)**Unix**.

To copy files from your host to a target machine:

1. In a file management utility on the host (e.g., Windows Explorer), select your files, then select **Copy** from the context menu.
2. In the left pane of the **Target File System Navigator**, expand the directory listings as needed under the entry for the target to which you're copying the files, right-click the destination directory, then select **Paste**.

You can also drag and drop files to a target. To do this:

- Drag your selected files from any program that supports drag-and-drop (e.g., Windows Explorer), then drop them in the appropriate destination directory shown in the **Target File System Navigator**.

Copying files from a target to the host

To copy files from a target machine to your host:

1. In the **Target File System Navigator**, expand the directory listings as needed under the entry for the target from which you're copying the files, select the files you want to copy, then right-click and choose **Copy to** > (and then)**File system**.

   The **Browse For Folder** dialog box appears.

   NOTE:

To import files directly into your workspace, select **Copy to** > (and then)**Workspace**. This brings up the **Select Target Folder** dialog box.

2. Select the destination directory and click **OK**.

You can also copy files to the host using drag-and-drop. To do this:

- Drag your selected files from the **Target File System Navigator** and drop them in the **Project Explorer**.

Page updated: August 11, 2025

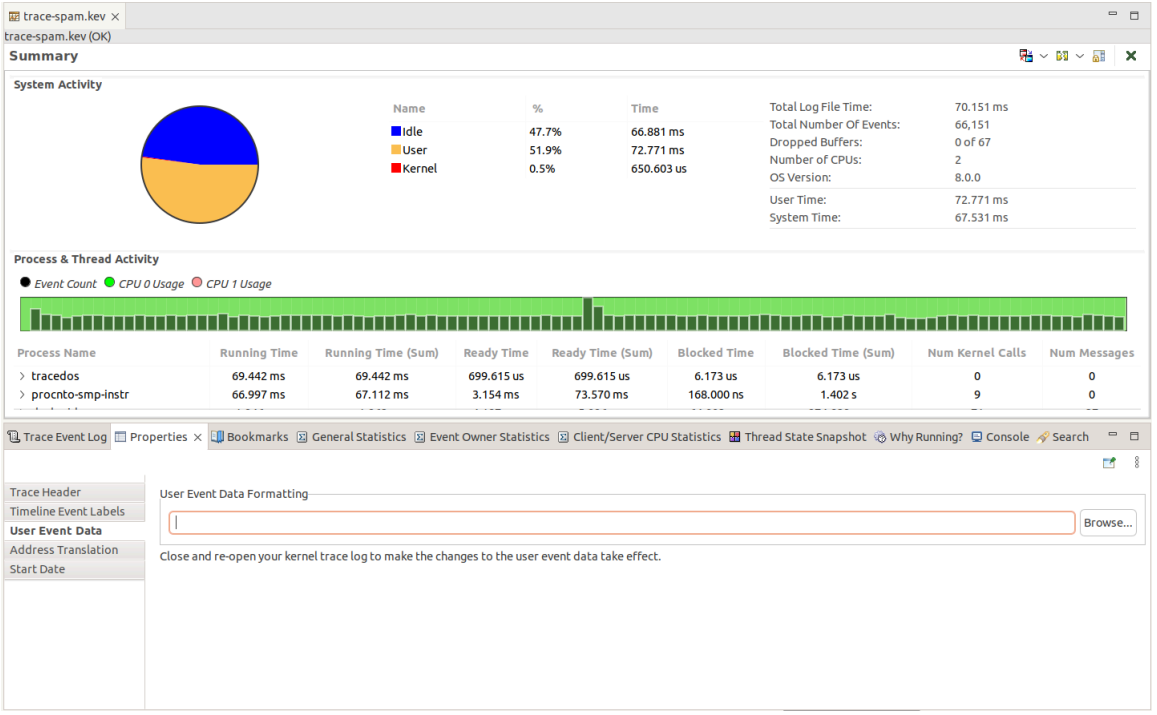
Formatting user events with the System Profiler

Using the "*TraceEvent API*", it's possible to emit custom trace events. However, these events are often difficult to understand in the Momentics IDE, since they don't have named fields the way QNX-provided events do. To address this, we've provided a mechanism for users to specify custom event formatted using an XML file. This guide provides an introduction to user event formatting so that you can interpret trace event data more effectively. The `datakey` specifier from the top-level `eventdefinitons` tag includes several subtags which are useful for formatting custom user events from the kernel. This data appears in the **Trace Event Log** table of the System Profiler. This guide covers the following subtags: `event`, `bitmask`, `condition`, and `enum`. For the XML reference, go to "*System Profiler: user event formatting XML reference*."

Overview

You can upload an XML file with your user formatting using the Properties view of an open trace. To upload the file:

1. Navigate to the **Properties** tab of an open trace.
2. Click the **Browse** button to locate the file with your XML formatting.



Enums

It's useful to use an `enum` to match against a number of expected values.

Matching values directly:

For user (class 6) event 10, you could match against an `enum` as follows:

```
<eventdefinitions>
    <datakey format="%x mydata">
        <event class="6" id="10" />
        <enum name="mydata" value="0x0" string="zero"/>
        <enum name="mydata" value="0x1" string="one"/>
        <enum name="mydata" value="0x2" string="two"/>
        <enum name="mydata" value="0x3" string="three"/>
        <enum name="mydata" value="0x4" string="four"/>
        <enum name="mydata" value="0x5" string="five"/>
        <enum name="mydata" value="0x6" string="six"/>
        <enum name="mydata" value="0x7" string="seven"/>
    </datakey>
</eventdefinitions>
```

In this case, the `enum` takes the value of the *mydata* field, and does a simple equality check against the values provided. It displays the provided string on a match.

For example, if the *mydata* field is equal to 5, then the `enum` outputs the following:

```
Event Data:         mydata:0x5 five
Event Data Hex:     0x05 0x00 0x00 0x00 0x00 0x00 0x00 0x00
```

Matching values after applying a mask:

You can match values after applying a mask to read flags. For user (class 6) event 8, you can apply the following mask:

```
<eventdefinitions>
    <datakey format="%x mydata">
        <event class="6" id="8"/>
        <enum name="mydata" mask="0x00000001" string="MASK1" />
        <enum name="mydata" mask="0x00000002" string="MASK2" />
        <enum name="mydata" mask="0x00000004" string="MASK3" />
        <enum name="mydata" mask="0x00000003" string="MASK5" />
        <enum name="mydata" mask="0x00000008" string="MASK8" />
    </datakey>
</eventdefinitions>
```

An `enum` mask works by applying the mask to the value read from the field in the format string and checking to see if the masked value is equal to the mask.

For example, if the *mydata* field is equal to 5, then the `enum` outputs the following:

```
Event Data:         mydata:0x5 MASK1 MASK3
Event Data Hex:     0x05 0x00 0x00 0x00 0x00 0x00 0x00 0x00
```

Bitmasks and Enums

If multiple values are combined into a single value, then you can apply a bitmask and an `enum` to extract data.

For example, a bitmask and an `enum` are used to match against the lower 3 bits of a value. For user (class 6) event 10, you could apply a mask as follows:

```
<eventdefinitions>
    <datakey format="%x enum_bitmask">
        <event class="6" id="10" />
        <bitmask value="0x00000007" />
        <enum name="enum_bitmask" value="0x0" string="zero"/>
        <enum name="enum_bitmask" value="0x1" string="one"/>
        <enum name="enum_bitmask" value="0x2" string="two"/>
        <enum name="enum_bitmask" value="0x3" string="three"/>
        <enum name="enum_bitmask" value="0x4" string="four"/>
        <enum name="enum_bitmask" value="0x5" string="five"/>
        <enum name="enum_bitmask" value="0x6" string="six"/>
        <enum name="enum_bitmask" value="0x7" string="seven"/>
    </datakey>
</eventdefinitions>
```

In this case, you apply the bitmask first, changing the value of *enum_bitmask*. Then, you apply any regular `enum` matching logic. For example, if the *mydata* field is equal to `0xFFFFFFFF`, then the formatting outputs the following:

```
Event Data:         enum_bitmask:0x7 seven
Event Data Hex:     0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF
```

Additionally, since this `datakey` only contains a single element (*enum_bitmask*), the name attribute is not strictly required on the `enum` tags:

```
<eventdefinitions>
    <datakey format="%x enum_bitmask">
        <event class="6" id="10" />
        <bitmask value="0x00000007" />
        <enum value="0x0" string="zero"/>
        <enum value="0x1" string="one"/>
        <enum value="0x2" string="two"/>
        <enum value="0x3" string="three"/>
        <enum value="0x4" string="four"/>
        <enum value="0x5" string="five"/>
        <enum value="0x6" string="six"/>
        <enum value="0x7" string="seven"/>
    </datakey>
</eventdefinitions>
```

Conditions

Conditions are used to conditionally apply user event data formatting. For example, you can use conditions to display detailed error codes (when the status code is incorrect).

```
<eventdefinitions>
    <datakey format="%2s1d status %4u1x pid" offset="0" wide_offset="0"
show_masked_value="false">
        <event class="6" id="12" />
        <enum name="status" value="0" string="OK" />
        <enum name="status" value="-1" string="Error" />
    </datakey>

    <datakey format="%4s1d error_no" offset="6">
        <event class="6" id="12" />
        <condition key="status" value="-1" />
        <enum name="error_no" value="1" string="EBADREQ" />
        <enum name="error_no" value="2" string="ENOWORKER" />
    </datakey>
</eventdefinitions>
```

The condition only applies if the status field is parsed as a negative value. If this is the case, then the condition adds the format ***error_no*** to the event data and applies the `enum`. For example:

```
Event Data:         status:-1 Error pid:0x7c5 error_no:1 EBADREQ
Event Data Hex:     0xFF 0xFF 0xC5 0x07 0x00 0x00 0x01 0x00
                    0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
                    0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
                    0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
```

The example uses offsets to specify where to find the ***error_no*** value. You specify the offset since you don't want to start at 0 (which is where the status field is encoded). The offset is an index into the array of event data. For example, you can find the ***error_no*** code by indexing the event hex data and reading 4 bytes at offset 6 (the size is specified by the format string).

```
Event Data Hex:
0    1    2    3    4    5    6    7
0xFF 0xFF 0xC5 0x07 0x00 0x00 0x01 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
```

NOTE:

The event data hex values are little endian in these examples. If you aren't familiar with little-endian encoding, then be aware that each piece of data might appear backwards. This is why the following set of bytes is interpreted as 1: `0x01 0x00 0x00 0x00`.

Using a mask and value in a condition

You can combine a mask and a value in a condition, which is useful when you want to locate multiple fields within data.

For example, consider the following example; it masks the header element and, if the value matches 0 after the mask, then it applies additional formatting. You could use this to match against the lower two bits of header field encoded with a payload type.

```
<eventdefinitions>
    <datakey format="%4u1x id %4u1d size %1s0 data" offset="10">
        <event class="6" id="12"/>
        <condition name="header" mask="0x3" value="0"/>
    </datakey>
</eventdefinitions>
```

The above formatting results in the following output:

```
Event Data:         status:0 OK pid:0x7c5 header:0x4 BROADCAST
id:0x9abccba9 size:14 data:exampleMessage
Event Data Hex:     0x00 0x00 0xC5 0x07 0x00 0x00 0x04 0x00
                    0x00 0x00 0xA9 0xCB 0xBC 0x9A 0x0E 0x00
                    0x00 0x00 0x65 0x78 0x61 0x6D 0x70 0x6C
                    0x65 0x4D 0x65 0x73 0x73 0x61 0x67 0x65
```

In the above example, the formatting is applied even though the header field's value is `0x4`. The formatting applies because the 4 is masked out by the condition's mask and the resulting value is equal to the condition's value of 0.

You can provide a different format when the masked condition value is different:

```

<eventdefinitions>
    <datakey format="%4u1x id %4u1d size %1s0 data" offset="10">
        <event class="6" id="12"/>
        <condition name="header" mask="0x3" value="0"/>
    </datakey>

    <datakey format="%4u1x id %4u1x data" offset="10">
        <event class="6" id="12"/>
        <condition key="header" mask="0x3" value="1"/>
    </datakey>
</eventdefinitions>
```

In this case, only the `id` and `data` fields are applied since the header is masked and the value equals 1 instead of 0:

```

Event Data:         status:0 OK pid:0x7c5 header:0x1 id:0xbeefbeef
data:18
Event Data Hex:     0x00 0x00 0xC5 0x07 0x00 0x00 0x01 0x00
                    0x00 0x00 0xEF 0xBE 0xEF 0xBE 0x12 0x00
                    0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
                    0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
```

NOTE:

The difference between the formats in the example is related to whether or not the `size` field is parsed from the data.

Page updated: August 11, 2025

Running a kernel event trace

QNX Tool Suite | Integrated Development Environment User's Guide | Developer | Setup

You can create log configurations to define kernel event trace parameters, then run one of these configurations to perform a trace. The generated log file contains data gathered from a target machine, including kernel events, CPU activity, and process and thread activity.

Before you can gather data from a QNX target, you must create a connection to it.

NOTE:
In this SDP release, the instrumented version of the kernel (**procnto*-instr**) is the only version that we ship. This component includes a small, efficient module that generates events based on process and thread activity such as kernel calls and context switches. So, you don't need to confirm that this process is running on the target, unlike in previous releases.

To run a kernel event trace from the launch bar:

1. In the Launch Configuration dropdown, select a log configuration.
   Log configurations for kernel event traces are indicated by the System Profiler icon (). You can create a log configuration by selecting **New Launch Configuration**, which opens a wizard that guides you through this task. If you have an existing configuration, skip to Step 3.
2. Follow the steps in the **New Launch Configuration** wizard:
   a. Select an initial launch mode of `Log`.
   b. Select a launch configuration type of `Neutrino Kernel Event Trace`.
   c. Set the log configuration properties as needed.
   d. When you're finished setting the properties, click **Finish** to exit the wizard.
   You can then select the new Launch Configuration entry for this new log configuration to continue.
3. In the Launch Target dropdown, select the target that you want to trace.
4. Click the Log button () on the left of the launch bar.
   The kernel on the target begins logging event data to the output named in the log configuration. The procedure is explained in "How kernel event tracing works". In the **Kernel trace** window, you can see the progress of the trace and choose to run it in the background.

When the trace finishes, the IDE asks you to open the kernel event log (**.kev**) file. If you select `Yes`, it then displays the file's data in the editor pane and switches to the QNX System Profiler perspective, which contains many views that let you interpret the trace data for analysis.

NOTE:
You can also use the logging button () in the toolbar (in the upper right area of the IDE) to start a new trace based on the selected log configuration. The dropdown button () next to this icon opens a menu that lists the recently used log configurations, the **Log With...** submenu (see the next subsection), the **Log Configurations** option (which opens a window listing all log configurations), and an option for defining favorite configurations.

Starting a kernel event trace from the **Target Navigator**

QNX Tool Suite | Integrated Development Environment User's Guide | Developer | Setup

You can also manage log configurations and start kernel event traces from the **Target Navigator** view.

The prerequisite of having an active target connection also applies for this usage scenario.

To run a kernel event trace from the **Target Navigator**:

1. In the **Target Navigator** view, select the target (or any process listed under it) that you want to trace, then right-click to display the context menu.
2. Point to **Log With...** to display the log configurations submenu.

This submenu lists the supported log types. You can click the **1 Kernel Event Trace** entry to bring up the **Kernel Event Trace** window, which lists the available log configurations for running kernel event traces. If you have an existing configuration that you want to use, skip to Step 3.

If you haven't created a kernel event trace configuration, the IDE opens the **Log Configurations** window and defines a default configuration with a name like `Kernel Log of target`.

> **NOTE:**
> Even if you click **Close** without changing any settings or clicking **Log** to run a trace, the IDE still saves this configuration and you will see it listed in the **Kernel Event Trace** window.

The **Log With...** submenu also contains the **Log Configurations...** option, which opens the window for managing log configurations. Here, the toolbar in the upper left corner lets you create, duplicate, and delete log configurations, as well as edit their property settings.

3. Start the kernel event trace.
    - If you're in the **Kernel Event Trace** window, select the configuration to use, then click **OK**.
    - If you're in the **Log Configurations** window, click **Log**.

    The kernel on the target begins logging event data to the output named in the configuration. The **Kernel trace** window shows the progress of the trace and allows you to run it in the background.

When the trace finishes, the IDE asks you to open the kernel event log (**.kev**) file. If you select `Yes`, it then displays the file's data in the editor pane and switches to the QNX System Profiler perspective, which contains many views that let you interpret the trace data for analysis.

The **Target Navigator** view also provides a logging button in the title bar (), which starts a new trace based on the selected log configuration. The dropdown button () next to this icon lists the recently used configurations, the **Log With...** submenu, and an option for defining favorites.

Page updated: August 11, 2025

Get to know Eclipse

QNX Tool Suite | Integrated Development Environment User's Guide | Developer | Setup

The IDE is based on the Eclipse platform but adds many features and tools for debugging, profiling, and monitoring applications on QNX-based systems.

The UI of our IDE varies significantly from that of Eclipse, notably with the IDE's launch bar at the top, which automates building, running, and stopping programs. When appropriate, we refer the reader to the Eclipse documentation for components that are part of the underlying platform and are presented in the same way in our IDE. But you should follow the steps in this guide on project creation, launching, and debugging instead of those in the Eclipse documentation.

If you want to learn about the Eclipse platform and its built-in features, you can read the Eclipse documentation included with the IDE:

- The *Workbench User Guide* describes the Workbench desktop environment and provides both basic tutorials and detailed instructions for many different tasks.
- The *C/C++ Development User Guide* has information about the C/C++ Development Toolkit (CDT) and the Eclipse features for creating, editing, and navigating C/C++ projects.
- The *Eclipse Marketplace User Guide* explains how to use the Eclipse Marketplace Client to find and install useful features.
- The *EGit Documentation* describes how to share your project to a Git repository and work on it with your team.

Page updated: August 11, 2025

Finding thread synchronization problems with Valgrind Helgrind

QNX Tool Suite | Integrated Development Environment User's Guide | Developer | Setup

Valgrind Helgrind looks for memory locations accessed by multiple POSIX threads (pthreads) but for which no synchronization mechanism (e.g., locking) is used consistently. Based on the Helgrind results, the IDE lists the threads and memory locations involved in any such situations.

NOTE:
All Valgrind tools can be loaded and run from the command line. However, using the IDE is more convenient because it automates much of the setup by setting Valgrind command options based on UI fields and by copying the analysis results into the host workspace.

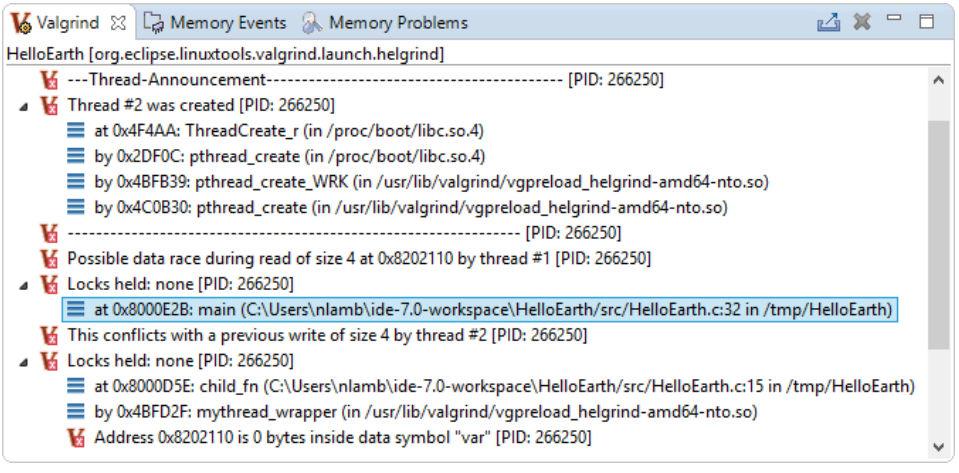
If your target image isn't configured to include the `valgrind` binaries and libraries, the IDE attempts to upload these components when you launch an application with a Valgrind tool enabled. For this to work, the target must have a writable filesystem.

To find thread synchronization problems with Valgrind Helgrind:

1. In the launch bar, expand the Launch Configuration dropdown (which is in the middle) and select the project in which you want to look for synchronization problems.
2. In the Launch Target dropdown (on the right), select the target for running your application.
3. In the Launch Mode dropdown (on the left), select Check.
4. Click the Edit button (⚙) on the right of the Launch Configuration dropdown.
5. In the configuration editor window, access the Valgrind controls by clicking the **Check** tab on the right and then the Valgrind radio button near the top of this tab.
6. Select **Helgrind** from the **Tool to run** dropdown.
7. **Optional:** You can change any settings to customize what gets reported in the Valgrind results.
   The **Helgrind** tab lets you set how much historical data is kept about conflicting memory accesses.
8. Click **OK** to save the configuration changes and close the window.
9. In the launch bar, click the Check button ( ).

The IDE switches to the QNX Analysis perspective. If necessary, the IDE first builds the binary before uploading it to the target. To analyze the application, the IDE instructs Valgrind to execute the uploaded binary with Helgrind instrumentation. Then, it creates a session for storing the Valgrind results; this new session is displayed in the **Analysis Sessions** view. When the program terminates, Valgrind writes the results to a log file, which the IDE copies into the directory for the new session.

The **Valgrind** view lists thread "announcements", which report the locations where threads were created, and details about all detected synchronization problems. These details include the threads involved in the conflicting accesses, the instruction addresses, source files, and lines of code where the conflicts occurred, and the related stack traces. The **Callers in stack trace** field in the **General Options** tab determines the depth of the displayed call chains:



You should see location information for functions in shared libraries. If you don't see this information, you must manually configure the loading of debug symbols.

If you double-click a trace line that has source file information, the IDE opens the source file at the indicated line. This feature lets you quickly find where memory shared between threads is accessed in an unsynchronized (and hence, dangerous) manner.

NOTE:

You can run multiple Valgrind sessions concurrently, using the same tool or different tools, on the same application or different applications. Valgrind log files always contain the PIDs of the Valgrind processes, so their names are always distinct.

Page updated: August 11, 2025

Configuring host-target communication

QNX Tool Suite | Integrated Development Environment User's Guide | Developer | Setup

The IDE on the host can communicate with target machines using an IP or a serial connection.

If you have a serial link, you can debug a program, but you need an IP link to use any advanced diagnostic tool integrated into the IDE. To enable IP communications, the target must run the `qconn` agent, which provides support, such as system profiling information, to IDE components on the host. For serial communications, the target must run the serial driver and `pdebug` support utility.

The sections immediately following explain how to configure your host and target to communicate with each other. When you then create a connection to the target, you must select a target type that reflects the communication method; more information is given "Supported target types".

Page updated: August 11, 2025

IP communication

Through an IP connection, the IDE can debug, monitor, and profile programs as they run on the target.

To set up an IP connection, you must first connect the target and host machines to the same network, and ensure that TCP/IP networking is functional on both machines. Then, you must configure your target as follows:

Launch the `qconn` agent on the target, from a command-line shell or the boot script.
By default, `qconn` runs on port 8000 (as indicated in its usage message). You can use another port by explicitly stating it in the command, as in: `qconn port=9000`
For details on the arguments and semantics for this agent, see the `qconn` entry in the *Utilities Reference*.

To support application debugging, monitoring, and profiling, you must verify on the target that:

- `qconn` is running (use `pidin | grep qconn`); this target agent must be listening on either an external port or a local port used with SSH tunnelling, which is explained in the next section
- `pdebug` is in **/usr/bin** and has execute permissions (`qconn` launches this service as required)
- the pseudo-terminal communications manager, `devc-pty`, is running (use `pidin | grep devc-pty`)

At this point, you can create a target connection in the IDE and specify the target's IP address and the port where `qconn` runs. Then, you can select that new target connection in the launch bar to tell the IDE that you want to run a program on that target machine. When you then launch a project, the IDE uses the configured IP link to upload the project's executable binary to the target and to talk to `qconn` and gather information about the program after it starts running.

CAUTION:
Don't run `qconn` in a production setting as part of the normal system setup. The `qconn` utility is highly insecure because it doesn't do user validation and always runs as root, which allows users to download and execute arbitrary code on the target. You can use the utility temporarily (e.g., if you're running an analysis tool that needs to send data to the host) by accessing the target machine through SSH and starting `qconn` manually. Or, you can secure `qconn` traffic using SSH, as explained below.

Page updated: August 11, 2025

Serial communication

A serial connection allows the IDE to debug a program but not to use any profiling or analysis tools. You generally set up this connection type only when TCP/IP networking isn't available on the target.

First, you must physically connect the host and target machines through a serial port. Then, you must configure your target to use the serial link:

1. If it's not already running, start the appropriate serial driver on the target.
   Typically, x86-based machines use the `devc-ser8250` driver. ARM-based machines use a board-specific driver; for details on the starting the right driver, consult your BSP User's Guide. When the serial driver is running, you can run `ls /dev/ser*` to see the path of the serial device.
2. Start the pseudo-terminal communications manager, `devc-pty`, by entering the following command:
   ```
   # devc-pty &
   ```
3. Determine the baud rate used by the serial port, using `stty` (this example assumes that you're using the first serial port):
   ```
   # stty </dev/ser1
   ```
   This command lists all parameters of the serial port. Look for the `baud=`*`baudrate`* entry; you'll need this information to properly configure both the target and host sides of the connection.
4. Start the debugging agent, `pdebug`, while specifying the previously reported baud rate:
   ```
   # pdebug /dev/ser1,115200 &
   ```
   This example uses a baud rate of 115200 but the `stty`-reported rate on your system may differ.

You can now use the newly enabled serial link to debug a program on the target. The link transports data between the debugger on the host (GDB) and the supporting agent on the target (`pdebug`).

Page updated: August 11, 2025

Securing qconn

By default, the traffic sent to `qconn` is unencrypted, which leaves it vulnerable to interception. You can encrypt this traffic by tunnelling it through `ssh`, which ensures that the traffic is secure.

To implement this security feature:

- The target has to have `sshd` installed and configured with either password authentication or public key authentication for the root user. You should therefore see files like the following on the target:
  /etc/ssh/ssh_host_rsa_key
  /etc/ssh/ssh_host_rsa_key.pub
  /etc/ssh/sshd_config

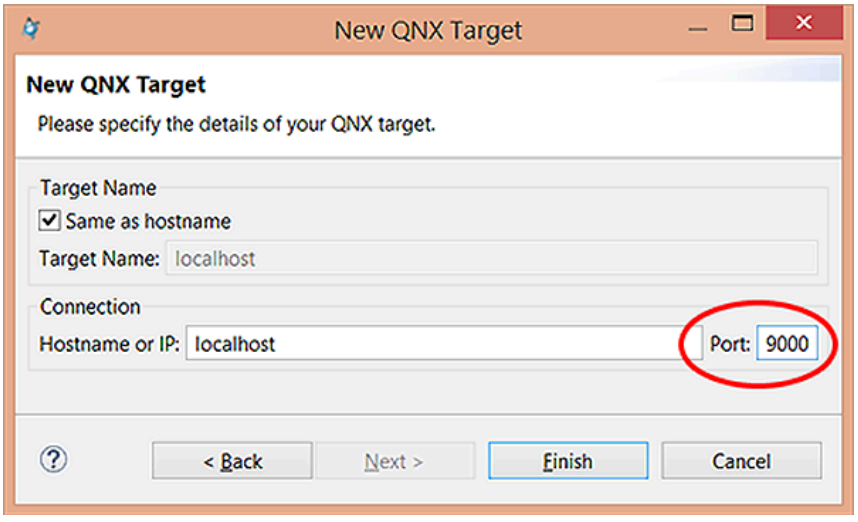
  You may see different files if you're using a different authentication protocol (e.g., DSA).
- The host has to have an SSH client.

To configure a secure connection between the host and target:

1. On the target, run `sshd`.
2. Start the `qconn` agent on the target, while specifying that it should accept connections only from the local host:
   ```
   qconn bind=127.0.0.1
   ```
3. On the host, establish an SSH tunnel by running `ssh` with the proper options:
   ```
   ssh qnxuser@target_host -N -L local_port:localhost:target_port
   ```
   where:
   - `-N` instructs `ssh` to not run a shell.
   - `-L local_port:localhost:target_port` specifies the local (host machine) port used for tunnelling (e.g., 9000), followed by the keyword `localhost` (alternatively, you could use the address 127.0.0.1), then the target port where `qconn` is running (usually 8000).

   NOTE:
   Depending on the SSH client program (e.g., PuTTY), you might have to manually specify the local and remote ports through UI fields instead of running the above command.

4. When creating a target connection in the IDE, instead of specifying the target's IP address and port, you must specify the local IP port used for SSH tunnelling, as shown below.



This setting makes the IDE connect to the target through the established SSH tunnel. The SSH service will forward all traffic from the local port to the target port.

Page updated: August 11, 2025

Working with QNX Momentics IDE

QNX Tool Suite | Integrated Development Environment User's Guide | Developer | Setup

QNX Momentics IDE provides a graphical environment for developing, running, debugging, and profiling applications.

The IDE interacts with the standard QNX build utilities and automates many build steps such as makefile generation. You can transparently access the remote machine on which your applications run because the IDE uploads any needed executables and libraries, and handles communication with the remote components that provide runtime data.

In addition to building and launching applications, the IDE can attach a debugger to a remote process. It also integrates third-party tools for analyzing the RTOS and active processes on the remote machine. These tools provide data about kernel activity and memory and resource usage that help you analyze performance at the application and system-wide level.

Page updated: August 11, 2025

Importing Memory Analysis data

You can import Memory Analysis sessions from XML files generated by the IDE or import logs written by the **librcheck** library as an application ran.

Typically, XML files containing session data are already on the host machine, because you obtain them from either the IDE or other team members who share analysis results. If you run an application from a command line that loads **librcheck** and redirects the event and trace output to a file, you can copy over the log file to the host by using the Target File System Navigator.

To import Memory Analysis data:

1. Select **File** > (and then)**Import** > (and then)**QNX** > (and then)**Memory Analysis Data**, then click **Next**.
   Alternatively, if the **Analysis Sessions** view is open, you can right-click anywhere in it and choose **Import Session**, select **Memory Analysis Data** from the wizard list, then click **Next**.
2. In the **Input File** field in the **Import Memory Analysis Data** window, specify the file containing the data that you're importing.

   You can enter an absolute host path or click **Browse** to open a window that lets you choose a workspace or local file, using a file selector. When providing a name, you can use *${workspace_loc:*}* to specify the current workspace; for example: **${workspace_loc:*project_name*/*results_file*}**.

   The filename must end with an **.xml** or **.rmat** extension. When importing logs (**.rmat** files), you must pick a session for storing the data and optionally, specify the paths of the binary files, by following the next three steps. For XML files, skip to Step 6.
3. Select a session for storing the log data.
   In this window, the Memory Analysis sessions are listed from oldest (at the top) to newest (at the bottom), with checkboxes that let you select just one session. Note that the session you select must be open, as indicated by the open session icon (). There's also a **Create New Session** button, if you want to store the log in its own session.
4. **Optional:** Click **Next** to proceed to the **Binary Path** dialog, then specify the executable binary and any shared libraries that ran during the analysis session.

   To see line numbers in the event data (including the stack traces), you must have copies of the executable binary and library files with debug symbols on your host machine, so the IDE can associate the events with call sequences.

   Next to the **Select executable file** text field, you can click **Browse** to open a popup window that lets you choose a workspace or local filesystem directory. The text field gets populated when you make your selection, for example: **${workspace_loc:HelloWorld/build/x86_64-debug/HelloEarth}**

   The **Libraries Search Path** panel lists the paths that the IDE should search to find shared libraries referenced by memory event data. This UI control behaves the same way as the **Libraries** tab in the launch configuration.
5. **Optional:** Specify the source code paths.
   The **Source Lookup Path** panel lists the paths that the IDE should search to find source code that's related to memory events but wasn't compiled on the host. This UI control behaves like the **Source** tab in the launch configuration, meaning you don't need to define these paths if your source code was built on the host. The difference is that you can enable recursive directory searching, by checking the boxes in the **Recurse** column.
6. Click **Finish**.

   The IDE begins importing the data.

   If you're importing a session from an XML file or you chose to create a session for storing an imported log, you'll see a new entry in **Analysis Sessions**. The session header begins with the `imported:` string but then follows the usual naming convention while using a new, unique session number. You can rename the session by right-clicking its entry and choosing **Rename**.

   If you're importing a log into an existing session, you'll see a new process entry in that session.

When the import operation finishes, you can view the results from the earlier analysis session. The Memory Analysis reference explains how to interpret the results.

Page updated: August 11, 2025

Importing Valgrind logs

You can import Valgrind logs from text files exported from the IDE or copied from a target where a Valgrind tool was run on an application.

If you run Valgrind through the command line instead of the IDE, you can copy the resulting log file from the target to the host by using the Target File System Navigator.

To import Valgrind logs:

1. Select **File** > (and then)**Import** > (and then)**QNX** > (and then)**Valgrind Log**, then click **Next**.
   Alternatively, if the **Analysis Sessions** view is open, you can right-click anywhere in it and choose **Import Session**, select **Valgrind Log** from the wizard list, then click **Next**.
2. In the **Import from File** field, specify the file containing the log that you're importing.

   You can enter an absolute host path or click **Browse** to open a popup window that lets you choose a workspace or local file, using a file selector.

   Memcheck and Helgrind each produce one log file named **valgrind_*pid*.txt**. So you should select a file with this type of name to import a log written by one of those tools. Massif and Cachegrind each produce two log files. One is named similarly to what's produced by the other two tools but contains simply information and warning messages from the Valgrind binary and no session data. You should instead select the other file, which has a name beginning with the tool name (e.g., **massif_*pid*.txt**), for importing data.

   NOTE:
   Importing Massif session data isn't fully supported at the present time. You can import a Massif log file but the data aren't graphed in the **Heap Chart** view like they are when you run the tool from the IDE. Instead, the raw text output is listed in the **Console** view and the heap trees from detailed snapshots are shown in a flat list in the **Valgrind** view; although, the heap tree button () to show the memory breakdown isn't available. Thus, it's possible to see but difficult to interpret the stack traces of the program's heap allocations.

   Cachegrind session data also aren't presented properly when imported. In this case, the **Valgrind** view doesn't work because nothing is shown, so you must read the raw text output in the **Console** view.
3. **Optional:** Click **Next** to proceed to the **Source Lookup** dialog, then specify the source code paths.
   The **Source Lookup Path** panel lists the paths that the IDE should search to find source code that's related to memory errors or heap data but wasn't compiled on the host. This UI control behaves the same as the **Source** tab in the launch configuration, meaning you don't need to define these paths if your source code was built on the host.
4. Click **Finish**.
   The IDE begins importing the log data into a new session. A new entry appears in **Analysis Sessions**, with the imported text file listed under the `Logs` heading. The session header contains the `Import` string followed by a new, unique session number.

When the import operation finishes, you can view the results from the earlier analysis session. The raw output from the analysis tool is displayed in the **Console** view, while a more readable format of the output is presented in the **Valgrind** view.

The following list tells you where to find information about interpreting the output for specific tools:

- Memcheck — Finding memory corruption with Valgrind Memcheck
- Massif — Analyzing heap memory usage with Valgrind Massif
- Helgrind — Finding thread synchronization problems with Valgrind Helgrind
- Cachegrind — Analyzing cache usage with Valgrind Cachegrind

Page updated: August 11, 2025

Integrated tools

The IDE works with the GNU Debugger (GDB) and many runtime analysis tools, some of which are made by QNX and others by a third party. Using these tools, the IDE can report memory errors, heap usage, and function runtimes, and display kernel event trace data and target state information.

System Information, GDB, and the System Profiler are helpful in all debugging use cases—improper program results, process hanging, and process crashing. The last two tools can be used concurrently with the memory-analyzing and profiling tools, which find more specific problems.

For debugging improper program results and crashes, the Memory Analysis, Valgrind Helgrind, and Valgrind Memcheck tools are useful. For detecting the cause of a process hanging, the Application Profiler and System Information are helpful.

NOTE:
Code Coverage isn't primarily used for debugging or optimization—its purpose is to measure the effectiveness of unit tests. Valgrind Cachegrind also isn't intended for debugging but instead helps you improve performance by optimizing cache usage.

Table 1. IDE tool support for debugging use cases

Tool	Debugging capabilities	Use cases	Advantages	Drawbacks
Application Profiler	When run in sampling mode, takes execution position snapshots (i.e., the current address being executed) at regular intervals. The execution positions provide a summary of where in the code the program is spending most of its time.	Process hanging	• tells you the threads where the program is getting stuck without requiring you to step through the code • sampling mode has low overhead, so the data aren't biased towards particular functions (e.g., short or frequently called functions)	• reveals limited information; the sampling data tell you on average how often certain threads run, but nothing about how they use memory or interact • inaccurate over short time intervals, because the sampling size is small
GDB	Lets you step through the code to see which paths are followed, how variable values change over time, and whether certain lines of code are reached. You can also debug core files, to see what a program was doing when it crashed.	Improper program results Process hanging Process crashing	• gives a "white box" view of the program's state as it runs so you can determine the exact code changes needed to fix any problems • works well with other tools because it stops execution when a crash occurs, which lets you run another tool • by loading a core file, you can work backwards from a crash to determine its cause, without tracing the program's entire execution	• requires you to have a good idea about which areas of code are causing problems (except with core files) • starting and stopping execution in the debugger changes thread timing, making some situations hard to reproduce • core files show the crash symptoms but don't always reveal their cause (e.g., errors can occur long after a bad memory allocation)

Tool	Debugging capabilities	Use cases	Advantages	Drawbacks
Memory Analysis	Tracks allocation and deallocation operations and provides data to the IDE so it can report memory leaks and other common errors such as buffer overruns.	Improper program results Process crashing	• doesn't require recompiling the program and is easily configured in the launch configuration • has much less overhead than other tools (i.e., only a 30% increase in memory usage and a performance cost of 3 to 5%)	• finds relatively few errors; for instance, a source pointer containing a bad address passed to *memcpy()* is reported as an error, but uninitialized memory isn't
System Information	This IDE perspective provides memory, CPU, and resource usage data about the processes on a target. It also displays thread states, so you can see if a process is hanging because one of its threads is blocked or busy.	Process hanging	• quick and easy to use; doesn't require any launch configuration or build support, and you can switch to this perspective and see the target information at any time	• tells you which threads are blocked or busy (and possibly causing a process to hang) but not why
System Profiler	Displays event data generated by the instrumented kernel running on a target. These data include process- and thread-level metrics on CPU consumption, execution times, and the number of messages sent, which gives you an idea of the event sequence behind a program failure.	Improper program results Process hanging Process crashing	• lets you find problems related to the interaction of many processes or threads, by providing data about message passing and other IPC mechanisms • CPU statistics can quickly tell you when a particular thread began monopolizing a CPU, which is often the cause of a process hanging • can be used concurrently with other analysis tools, because you can start a kernel event trace no matter which processes and tools are running	• less precise way of debugging—it's better to debug a core file or trace the program's execution with GDB to find where in the code a problem occurs • even short kernel event traces that last only a few seconds produce a lot of data, making it hard to find insightful information; determining how to best filter the data can be complex

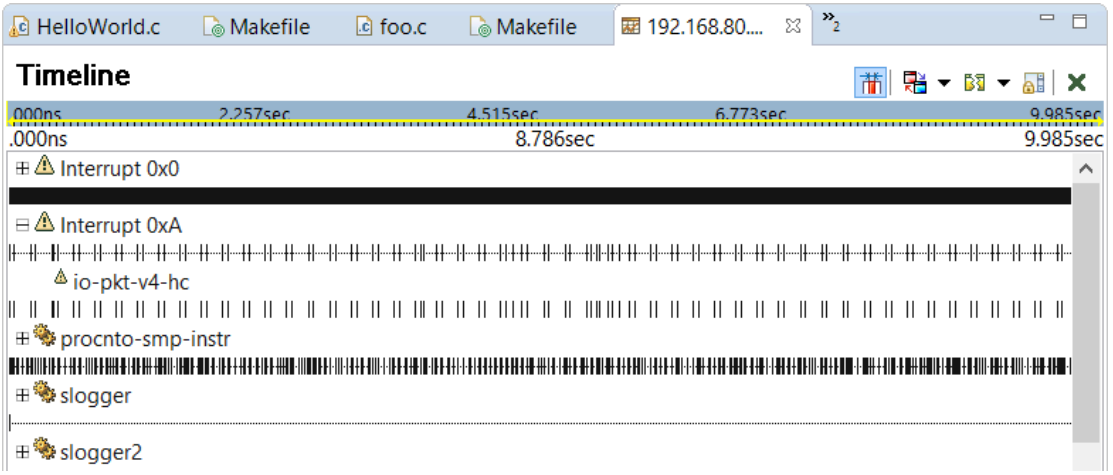
Tool	Debugging capabilities	Use cases	Advantages	Drawbacks
Valgrind Helgrind	Finds thread synchronization problems in programs that use pthreads, by detecting memory locations accessed by more than one thread but without proper locking or synchronization.	Improper program results Process crashing	• detects multiple classes of synchronization problems: POSIX API misuses, lock ordering problems that can cause deadlock, and data races; this helps you fix hard-to-find timing bugs and the program's design	• performance can be very poor; slowdowns up to 100 times aren't unusual • in the present version, for lock order errors, the tool prints only 2 cycles instead of the complete lock cycle
Valgrind Massif	Takes regular heap snapshots and outputs data describing heap usage over time and where most memory is being allocated. These data let you see which functions use too much memory as well as the heap contents any time the program crashed.	Improper program results Process crashing	• helps you find both design and implementation bugs, by highlighting memory gluttons as well as leaks	• sends session data to IDE only upon program termination, so you can't monitor heap consumption in real time • imposes significant overhead on programs, making them run about 20 times slower than usual
Valgrind Memcheck	Detects memory management problems by checking all reads and writes of memory. This tool finds memory leaks, bad frees, overlapping blocks in source and destination pointers, use of uninitialized values, and accesses to improper memory regions.	Improper program results Process crashing	• very precise; can detect invalid memory accesses at the byte level and uses of uninitialized memory at the bit level • informative; reports errors as soon as they're detected, giving the source line numbers at which they occurred and a backtrace of the current position	• significant performance impact; instrumented programs run at one tenth normal speed (or slower) and use two to three times more memory • can't detect some out-of-bounds memory accesses (e.g., if the first access to an array is beyond the last element)

Page updated: August 11, 2025

Examining interrupt handling times

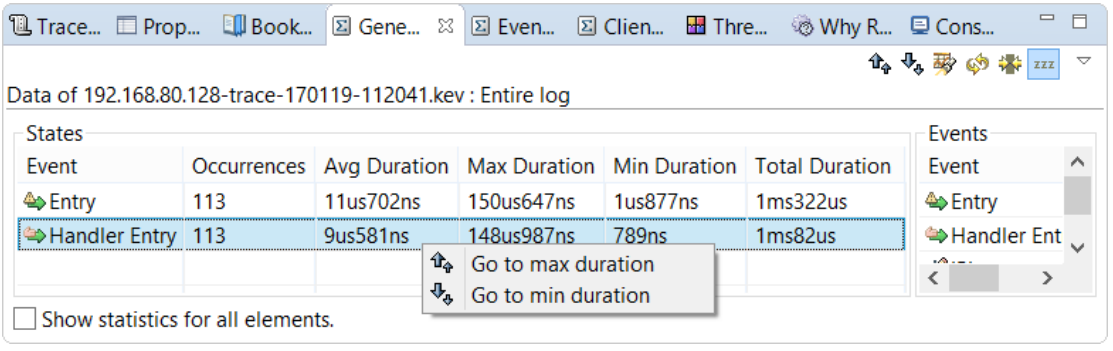
In realtime systems, it's crucial to minimize the time from the occurrence of a hardware event to the execution of code that handles it. The faster a system handles events, the more time it has to meet its deadlines. The System Profiler displays a timeline of events so you can precisely measure the times for various interrupt handling activities.

To see the sequence of events captured by the kernel event trace, click the Switch Pane dropdown in the editor controls (▾) and select **Timeline**. The Timeline pane illustrates the precise timing of events generated by all owners (sources).

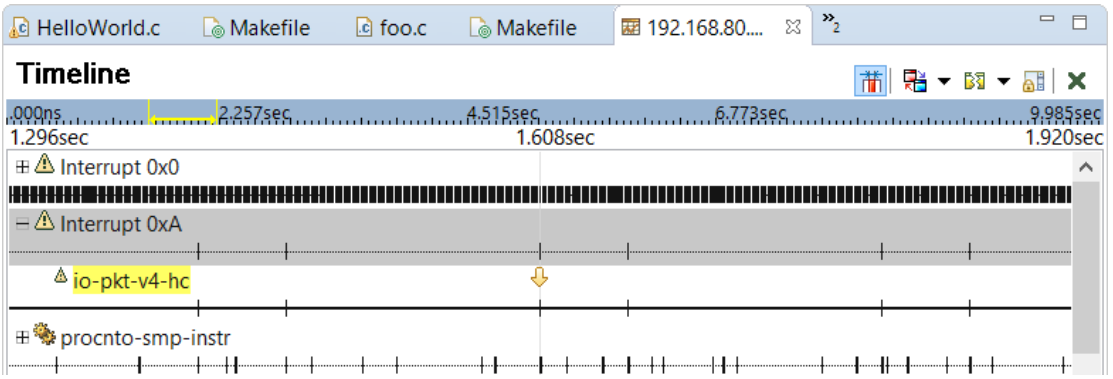


For each owner, the System Profiler draws a timeline, with individual events represented by vertical tick marks. Interrupts are listed at the top and their timelines show interrupt entry, exit, and deliver events, which indicate the start and end times of the microkernel's processing of individual interrupts. If you click the plus sign (+) next to an interrupt name, the System Profiler draws timelines that show interrupt handler entry and exit events, for all attached handler functions (just below the interrupt's timeline). Above each handler timeline, the name of the process in which the function runs is given.

To see data for events related to a particular interrupt vector, click its name in the editor display. The name and associated timeline get highlighted in grey. In the **General Statistics** view, which is displayed below the editor window, you can uncheck the **Show statistics for all elements** box to see aggregate data about the events for the selected owners only (i.e., the interrupt itself and any handlers).



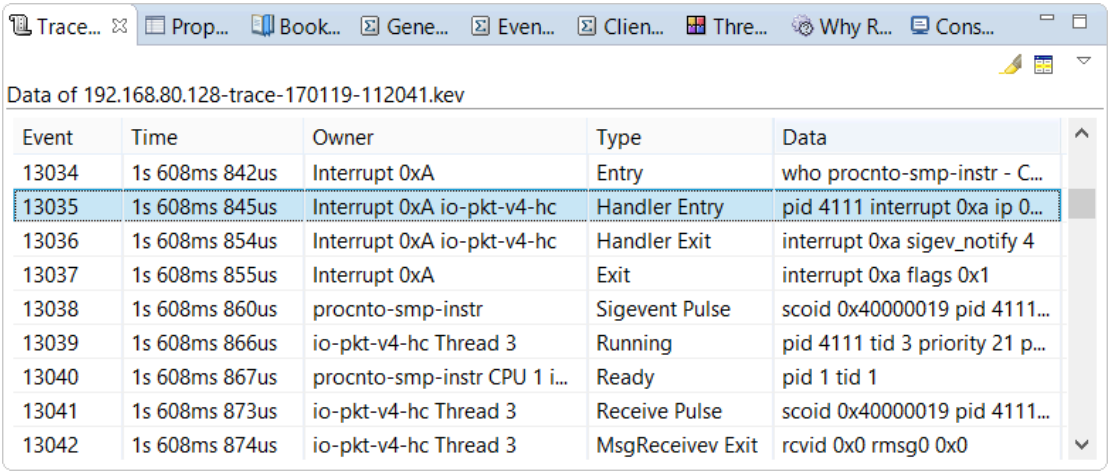
In the States table, the data reveal the number of events that marked the entry into a given state, and the average, maximum, minimum, and total times that an owner remained in that state. You can right-click in a table row and from the context menu, navigate directly to the events that mark the beginning of the maximum and minimum times in that state.



In this latest screenshot, the user has selected the `Interrupt 0xA` interrupt and navigated to a Handler Entry event owned by `io-pkt-v4-hc`. The toolbar in the upper left corner of the IDE provides buttons for navigating between events in the timeline. Because interrupt processing time is very short compared to the time period of most kernel event traces, the timeline display must be scaled, using the zoom feature, to easily distinguish events.

> **NOTE:**
> Some owners generate events so frequently (e.g., `procnto-smp-instr`) that you might accidentally keep clicking their events when trying to click events from other owners, even after zooming in considerably. To avoid this irritation, you can filter the timeline display to hide the timelines of highly active owners.

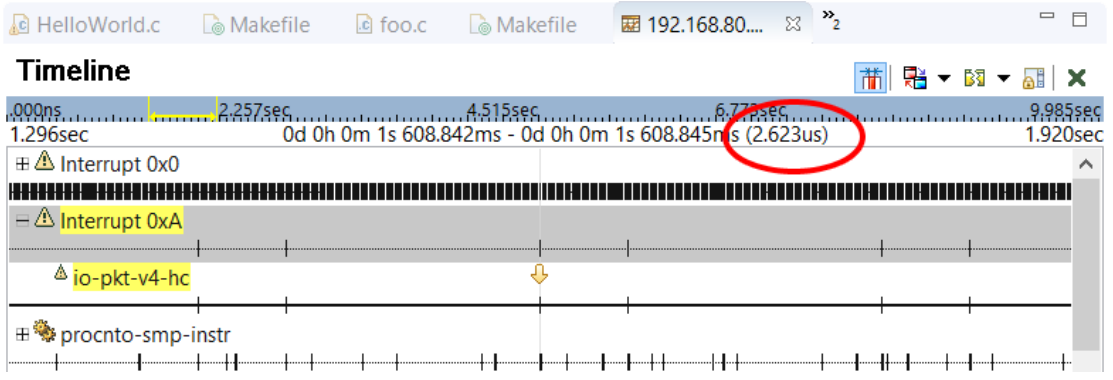
At this level of granularity, it's helpful to see individual events through the **Trace Event Log** view, which is also displayed at the bottom. This view is synchronized with the Timeline pane; selecting an event in one display navigates to that event in the other display.



Event	Time	Owner	Type	Data
13034	1s 608ms 842us	Interrupt 0xA	Entry	who procnto-smp-instr - C...
13035	1s 608ms 845us	Interrupt 0xA io-pkt-v4-hc	Handler Entry	pid 4111 interrupt 0xa ip 0...
13036	1s 608ms 854us	Interrupt 0xA io-pkt-v4-hc	Handler Exit	interrupt 0xa sigev_notify 4
13037	1s 608ms 855us	Interrupt 0xA	Exit	interrupt 0xa flags 0x1
13038	1s 608ms 860us	procnto-smp-instr	Sigevent Pulse	scoid 0x40000019 pid 4111...
13039	1s 608ms 866us	io-pkt-v4-hc Thread 3	Running	pid 4111 tid 3 priority 21 p...
13040	1s 608ms 867us	procnto-smp-instr CPU 1 i...	Ready	pid 1 tid 1
13041	1s 608ms 873us	io-pkt-v4-hc Thread 3	Receive Pulse	scoid 0x40000019 pid 4111...
13042	1s 608ms 874us	io-pkt-v4-hc Thread 3	MsgReceivev Exit	rcvid 0x0 rmsg0 0x0

Here, we can see that the selected Handler Entry event for the `io-pkt-v4-hc` function occurs just after the Entry event following the actual interrupt on vector `0xA`. The microkernel generates this first event when it starts processing the interrupt.

You can select both events (by holding **Ctrl** as you click each row) and see the brief time range displayed at the top of the Timeline pane. In this case, we've measured the kernel's *interrupt latency*, which is the time from when it starts processing the interrupt to when the handler starts running.



You can click the Handler Entry and Handler Exit events to measure the interrupt handler's runtime, which is just 9 us. After the handler exits and the kernel finishes processing the interrupt (as marked by the Exit event), we can see a Sigevent Pulse event, which shows that the handler has returned a pulse. This pulse evidently triggers the

networking stack process, because one of its threads gets scheduled to receive the pulse (as indicated by the sequence of Running, Receive Pulse, and MsgReceiveV Exit events for `io-pkt-v4-hc Thread 3`).

To check the runtime of an interrupt service thread, view the runtime of the thread registered to handle the interrupt.

Selecting the Handler Exit and MsgReceiveV Exit events lets you measure *scheduling latency*, which is the time from the last instruction in the handler function to the start of a response by a user (i.e., non-handler) thread. For more information about interrupt and scheduling latency, see the "Interrupt handling" section of the *System Architecture* guide.

Selecting the Entry and MsgReceiveV Exit events allows you to measure the end-to-end latency from the start of interrupt processing by the kernel to the response of a user thread. Here, that latency is 31 us, which proves that the interrupt handler runtime of 9 us is a relatively small part of the overall processing time. Thus, there is noticeable time needed for scheduling a thread to run and for it to receive the pulse returned by the handler.

Alternative measurements could include:

- How long it takes for the user thread to be scheduled rather than to receive the pulse, after the handler exits. You would look for either a Ready or Running event for one of the receiving process's threads, following the Handler Exit event. The time for scheduling may differ significantly from the time for pulse processing to begin if your target system runs many threads of distinct priorities.
- How long it takes for the user thread to either be scheduled or begin reacting to the pulse, after a specific handler exits. This is relevant when many handlers are attached to an interrupt vector (which is allowed, as explained in the *System Architecture* guide). If multiple handlers are run, this might skew the time before an application starts handling a pulse generated by one of them.

Page updated: August 11, 2025

Investigating performance bottlenecks

The profiling results tell you which functions are called frequently and which ones have the longest execution times. You can then analyze the target system or measure other aspects of an application's performance, to learn why certain functions are performance bottlenecks and to get ideas about how to optimize them.

Some common causes of bad performance are:

- Heavy heap usage — If a function allocates and frees a lot of heap memory many times, this slows execution because of the overhead of heap operations. You can analyze the program's heap usage to see which areas of code do this.
- Impact of other processes — On the target machine, other processes could be negatively impacting the performance of a given application. You can run a kernel event trace and examine the System Profiler results to see if problems such as starvation or priority inversion may be slowing the application at a particular point.
- Inefficient cache usage — If an application is doing data-intensive work but most of the working set isn't kept in cache memory, the frequent accesses to slower memory can noticeably reduce performance. Valgrind Cachegrind profiles a process's cache usage and reports statistics for each function and line of code, so you can pinpoint the exact sources of inefficient cache usage.

Page updated: August 11, 2025

Isolating client and server CPU loads

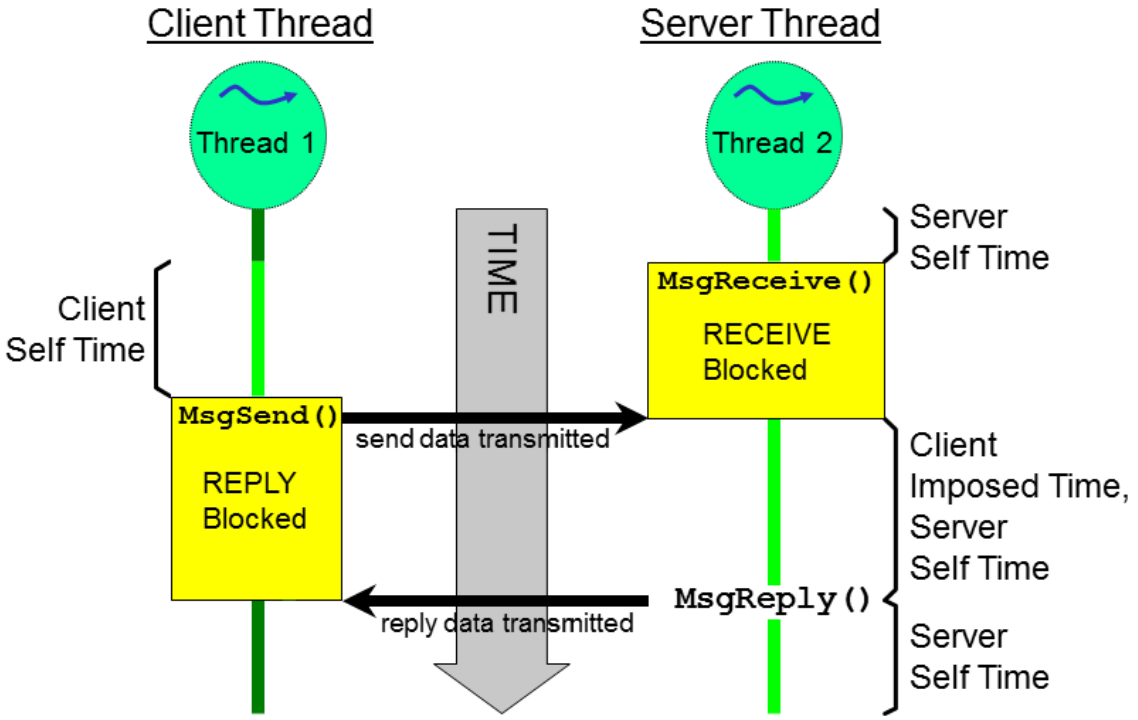
Heavy CPU usage can often be traced to server activity. Sometimes, a server doesn't need to be optimized but its client activity needs to be examined closely to see how much server time is accumulating based on client requests.

The primary IPC mechanism in QNX OS systems is message passing. The System Profiler uses message-passing events in the kernel event trace log to identify clients and servers when measuring their CPU times. Threads in client processes send messages to threads in server processes to ask them to perform work on their behalf. To understand why a target machine might be heavily loaded, you therefore need statistics on how much server work is imposed by individual clients.

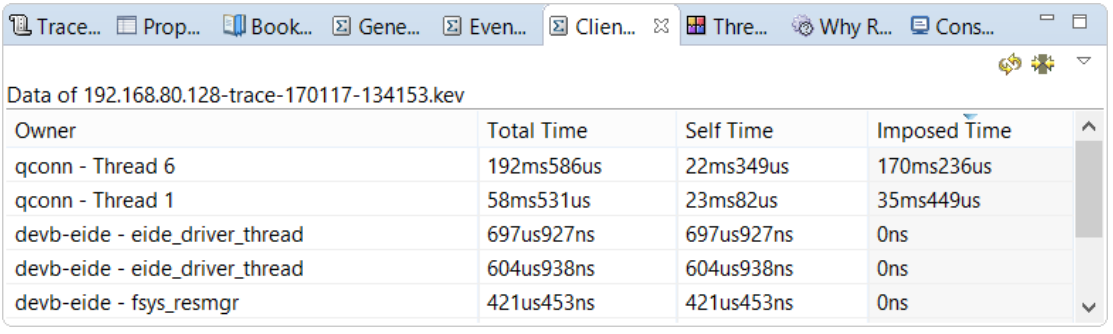
We use the term *self time* to mean that a thread is directly executing (i.e., doing something for itself). This applies to clients and servers. *Imposed time* means that a thread is indirectly executing (i.e., another thread is doing something for it). This applies only to clients, because servers sometimes execute on their behalf.

The following diagram illustrates the periods of self time versus imposed time for a basic client-server interaction:

Figure 1Self time versus imposed time based on a client-server message exchange



You can isolate the self and imposed CPU times by examining the **Client/Server CPU Statistics** view. Note that this view is displayed by the QNX System Profiler perspective, so if necessary, you should switch to this perspective when a kernel event trace finishes or you manually open a log file. To populate the view with statistics, you must click the **Gather statistics** button ( ).

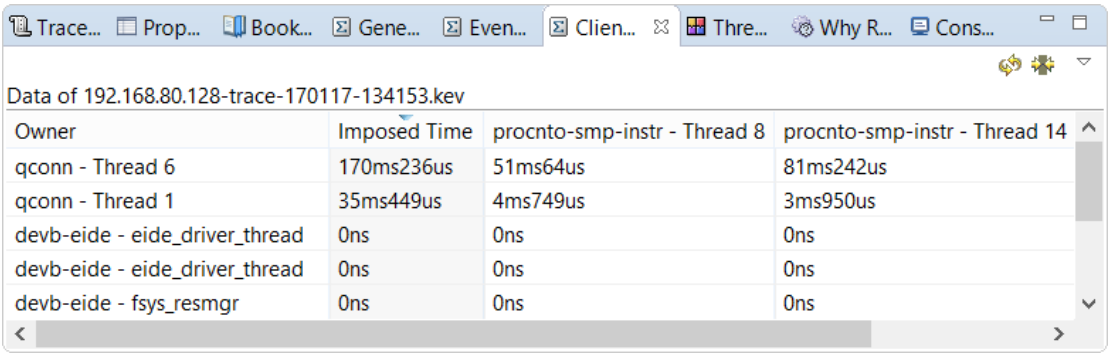


Owner	Total Time	Self Time	Imposed Time
qconn - Thread 6	192ms586us	22ms349us	170ms236us
qconn - Thread 1	58ms531us	23ms82us	35ms449us
devb-eide - eide_driver_thread	697us927ns	697us927ns	0ns
devb-eide - eide_driver_thread	604us938ns	604us938ns	0ns
devb-eide - fsys_resmgr	421us453ns	421us453ns	0ns

You can use the **Toggle global/selection statistics gathering** button ( ) to show time values based on the entire trace period or the selected timeframe only. This way, you can find which clients create the most work for servers. To easily spot these clients, click the **Imposed Time** header once to sort the table by this metric in descending order.

In this example, we can see that the `qconn Thread 6` thread has the longest imposed time. The next step is to determine which server threads are running the longest based on this client's requests. To do this, in the upper

right dropdown, you can click **Show all times** to display columns containing the imposed times for all servers.



Owner	Imposed Time	procnto-smp-instr - Thread 8	procnto-smp-instr - Thread 14
qconn - Thread 6	170ms236us	51ms64us	81ms242us
qconn - Thread 1	35ms449us	4ms749us	3ms950us
devb-eide - eide_driver_thread	0ns	0ns	0ns
devb-eide - eide_driver_thread	0ns	0ns	0ns
devb-eide - fsys_resmgr	0ns	0ns	0ns

Here, `qconn Thread 6` is imposing a heavy load on `procnto-smp-instr Thread 8` and `procnto-smp-instr Thread 14`. This makes sense because `qconn` must communicate with the kernel to extract information about the system state and all proceses and threads. If we were to examine the **Total Time** values (hidden in this screenshot) for these same kernel threads, we would find they're less than that for the `qconn` thread. This is because the QNX OS microkernel is a pure server process, meaning it consumes no CPU resources without being asked to do some work.

When you've identified the clients with the longest imposed times, you can redesign the affected applications to reduce the number of requests sent or even the number of client threads. This could involve, for example, making clients store copies of data locally to avoid querying servers as often.

Page updated: August 11, 2025

Building applications and an OS image for TRACE32-ICD debugging

QNX Tool Suite | Integrated Development Environment User's Guide | Developer | Setup

The TRACE32-ICD needs debug symbols to debug an OS image and the applications within it. To provide these symbols, you must modify the buildfile of your QNX System Builder project to include the debug versions of application binaries and to create symbol files for the `startup` script and QNX OS microkernel, `procnto`.

To prepare a QNX OS image for TRACE32-ICD debugging:

1. Build debug versions of any applications that you plan to debug.

   From the launch bar, you can select an application project in the launch configuration dropdown, choose either Run or Debug as the launch mode, then click Build ( ).

   From the **Project Explorer**, you can right-click a project and choose **Build Configurations** > (and then)**Set Active** > (and then)***arch*-debug** (where *arch* is the target architecture, such as x86_64), then right-click again and choose **Build Project**.

   You should now see the application binaries in the **build/*arch*-debug** subdirectories of the projects.

2. Add the binaries to the buildfile in the QNX System Builder project that defines your OS image.

   For each binary, you must list the target path for storing it, followed by an equal sign (=) and its workspace (host) path. You can use environment variables, such as *PWD*, in the workspace paths.

   *build

   ```
   ###############################################################################
   ## APPLICATIONS
   ###############################################################################

   /examples/HelloWorld=${PWD}../HelloWorld/build/${PLATFORM}-debug/HelloWorld

   ###############################################################################
   ## END OF BUILD SCRIPT
   ###############################################################################
   ```

   More information about buildfile syntax and adding content to images is given in the OS Image Buildfiles chapter of *Building Embedded Systems*.

3. Request the creation of symbol files for `startup` and `procnto`, by adding the `[+keeplinked]` attribute near the top of the buildfile.

   *build

   ```
   [+keeplinked]
   [image=0x10800000]
   # For u-boot, IFS image should be uncompressed
   [virtual=armle-v7,raw] .bootstrap = {
   # For IPL, IFS image should be compressed
   #[virtual=armle-v7,raw +compress] .bootstrap = {

       # Startup parameters:
   ```

   With this attribute enabled, the **mkifs** or **mkefs** utility invoked by the IDE creates **.sym** files for the two components when building the image.

4. Disable any watchdogs for the startup script or microkernel. This entails removing the watchdog-enabling options and commenting out the `wdtkick` command.

   *build

   ```
   [virtual=armle-v7,raw] .bootstrap = {
   # For IPL, IFS image should be compressed
   #[virtual=armle-v7,raw +compress] .bootstrap = {

       # Startup parameters:
       # '-W' -> enable watchdog (wdtkick should be uncommented when using this option)

       # startup-imx6x-sabreARD -n1 -m -W -s
       startup-imx6x-sabreARD -n1 -m -s

       procnto-smp-instr
   }
   [+script] .script = {

       ########################################################################
       ## WatchDog utility
       ## If startup is given '-W' parameter then the 'wdtkick' utility MUST
       ## be uncommented below.
       ########################################################################
       # display_msg Starting watchdog...
       # wdtkick
   ```

5. Build the image by right-clicking the project entry and choosing **Build Project**.

The **images** subdirectory then contains the IFS and symbol files.

Page updated: August 11, 2025

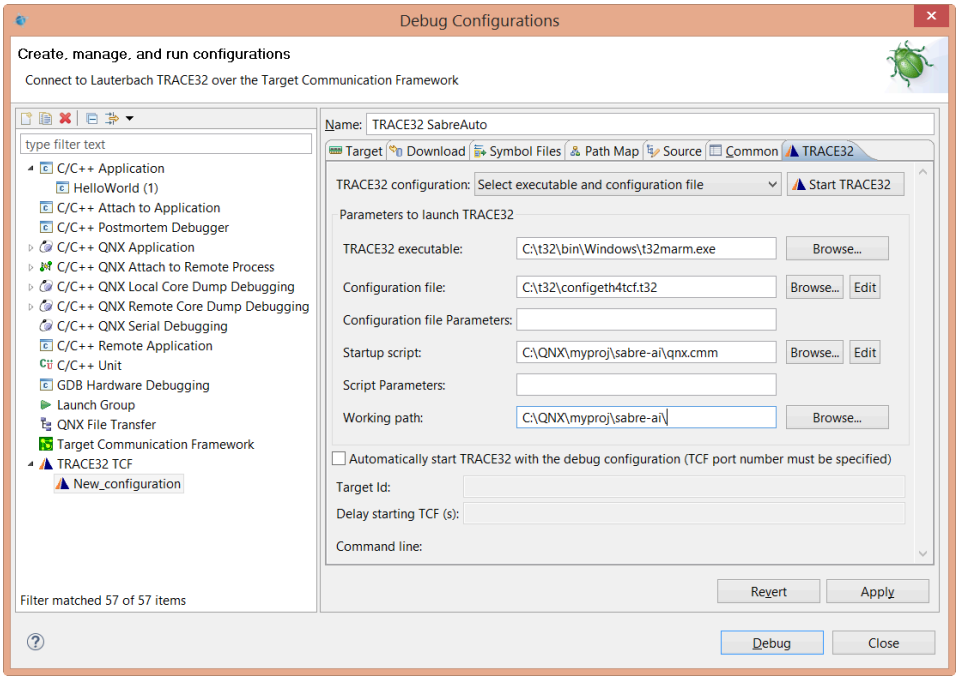
Creating a launch configuration for debugging an OS image

When you install the TRACE32 TCF plugin in the IDE, a new launch configuration type is added. You can then create a configuration of this type to use the TRACE32-ICD on a QNX OS image.

To define a configuration that supports debugging through the TRACE32 software:

1. Select **Run** > (and then)**Debug Configurations**.
2. In the list on the left, double-click **TRACE32 TCF** to create a new configuration.

   The configuration tabs appear in the main area of the window.
3. Provide a unique name in the **Name** text field at the top.
4. Click the **TRACE32** tab on the right to access the fields for configuring the TRACE32 software.
5. Follow the instructions in the "Option B: Select Executable and Configuration File" section of the *TRACE32 as TCF Agent* Lauterbach document (**app_tcf_setup.pdf**) found on the TRACE32 download page, to set the paths of the TRACE32 executable and startup script and other important fields.



6. In the configuration file, ensure that TCF is enabled. The file must contain a line with the text "`TCF=`" surrounded by empty lines.
7. In the **Debug Configurations** window, click **Apply** to save the settings, then click **Close**.

Page updated: August 11, 2025

Creating a TRACE32 startup script

To work with the TRACE32-ICD, you need to configure it for your specific hardware and OS image. You can automate the configuration by creating a startup script.

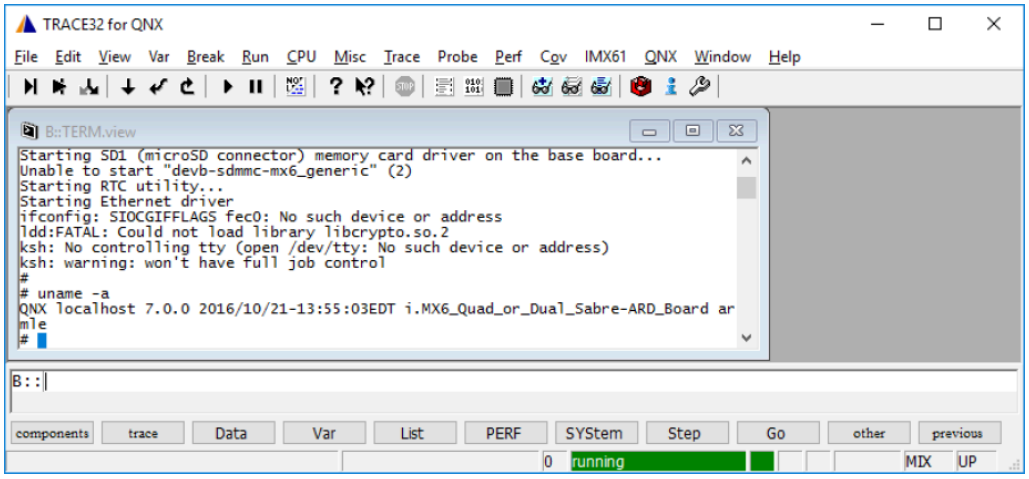
You may use the TRACE32 debugger to initialize the target hardware, download the QNX IFS onto the target, and start the image. Or, you may start your target hardware with the newly created image as usual (e.g., by booting from Flash), then attach the debugger to the running image. In either case, the debugger setup is done by a script that runs in the TRACE32 software on the host.

The writing of such a script depends heavily on the actual hardware and your preferred methodology and hence, is not covered here. For hardware setup, refer to the architecture-specific Lauterbach documentation; for configuring the debugger for QNX, refer to the *RTOS Debugger for QNX – Stop Mode* Lauterbach document (**rtos_qnx_stop.pdf**) found on the TRACE32 download page. Some sample scripts are found in ***install_dir*/demo/*arch*/kernel/qnx**.

When you've developed the startup script, you can prepare for debugging as follows:

1. Power up the TRACE32 hardware.
2. Start the TRACE32 software.
3. Power up the target hardware.
4. Run the script.

QNX OS should now be running on the target and the debugger should be attached to it.



You can now debug your OS image and its applications through the TRACE32 software, as explained in *RTOS Debugger for QNX – Stop Mode*. The next section explains how to debug through the IDE.

Page updated: August 11, 2025

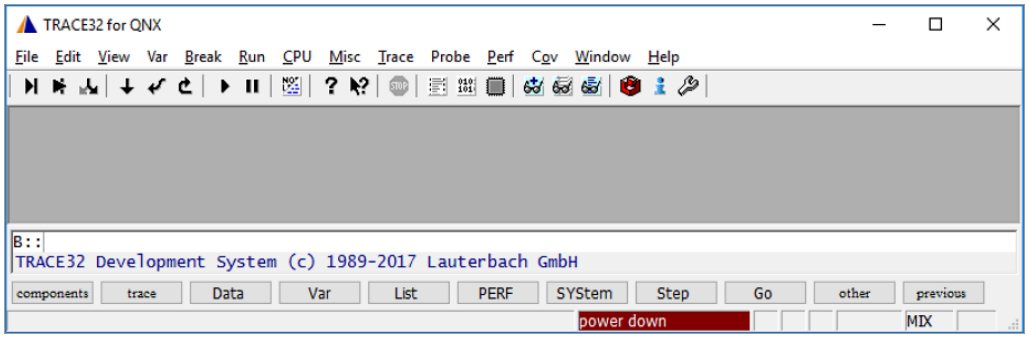
Installing the Lauterbach TRACE32 In-Circuit Debugger software

QNX Tool Suite | Integrated Development Environment User's Guide | Developer | Setup

To use the TRACE32-ICD either stand-alone or as a TCF agent, you have to install the appropriate software on your host. It's recommended to use the latest TRACE32 software version. For TCF, use at least the February 2016 version.

Follow the Lauterbach document entitled *ICD Quick Installation* (**icd_quick_installation.pdf**) to install the software. The exact steps depend on your host OS, power debug and power trace modules, and target board. You can find this document on the TRACE32 download page, under the TRACE32 Debugger Getting Started heading.

After installation, connect the TRACE32-ICD module to your host machine. Do not yet connect the JTAG cable to the target. Instead, power up the ICD module and start the TRACE32 software. When the software starts successfully, it shows a `power down` message in the status line. This message refers to the target, which isn't connected.



Close the TRACE32 software and power down the ICD module. Now you can connect the JTAG cable to your target.

For more information about the TRACE32-ICD, see the *ICD Debugger User's Guide* (**debugger_user.pdf**), which is included in the software package and copied (along with other Lauterbach documents) into ***install_dir*/pdf**. On Windows, the default installation directory is **C:\T32**; on Linux, there's no default directory but we recommend installing it under **/home/*user*/t32**. The document is also available on the TRACE32 download page, under the ICD In-Circuit Debugger heading.

Page updated: August 11, 2025

Installing the Target Communication Framework (TCF) in the IDE

QNX Tool Suite | Integrated Development Environment User's Guide | Developer | Setup

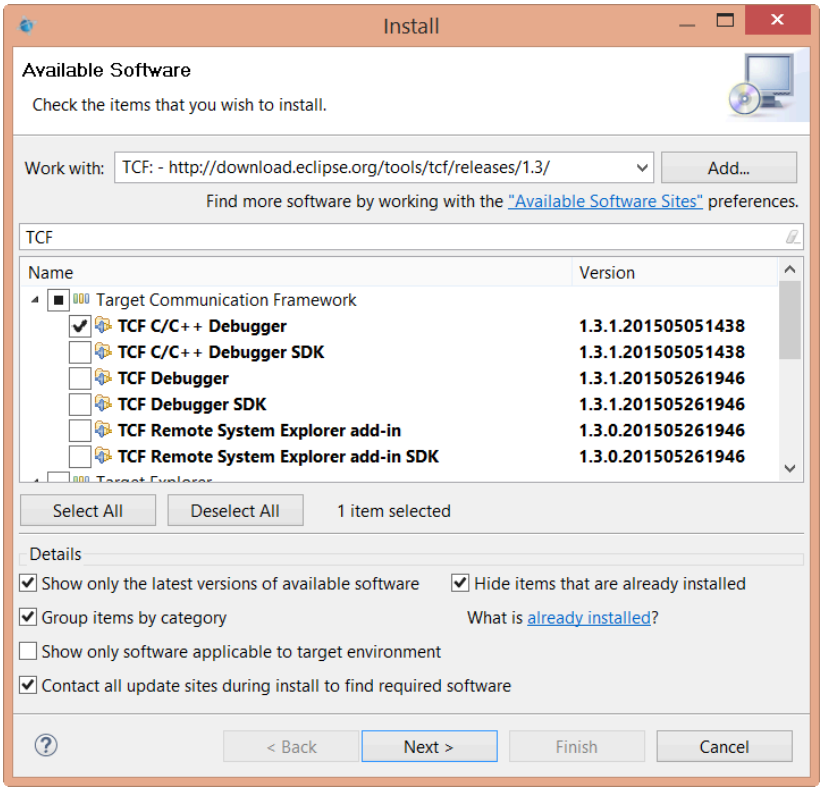
The TCF is an Eclipse framework designed for simplifying communication between development tools and embedded devices. This extensible, vendor-agnostic framework allows you to use the TRACE32-ICD through the IDE to debug QNX OS targets.

The TRACE32-ICD can be configured as a TCF agent. A TCF agent offers services to client tools by exposing commands and events. Through the TCF, the IDE can use the debug functionality of the TRACE32 software. Note that simultaneous usage of the TRACE32 UI and the IDE UI is also possible.

To install the TCF in the IDE:

1. In the IDE, select **Help** > (and then)**Install New Software**.
2. Click the **Add** button in the upper right corner of the **Install** window, next to **Work with**.
3. In the **Name** field of the **Add Repository** window, enter `TCF:`
4. In **Location**, enter `http://download.eclipse.org/tools/tcf/releases/1.3/`
5. Click **OK** to close the popup window.

   The list area of the **Install** window now shows entries for the software available from this location.
6. In the search field just above the list area, enter `TCF` to filter the view.
7. In the search results, check the box next to `TCF C/C++ Debugger`.



8. Click **Next** twice, to advance to the **Review Licenses** dialog.
9. Accept the license agreement, then click **Finish**.

   The window closes and the IDE begins downloading the TCF debugger package. Progress is displayed in the bottom right corner.
10. When the download finishes and the IDE prompts you to restart Momentics, click **Yes** to complete the installation.

Page updated: August 11, 2025

Installing the TRACE32 TCF Eclipse plugin

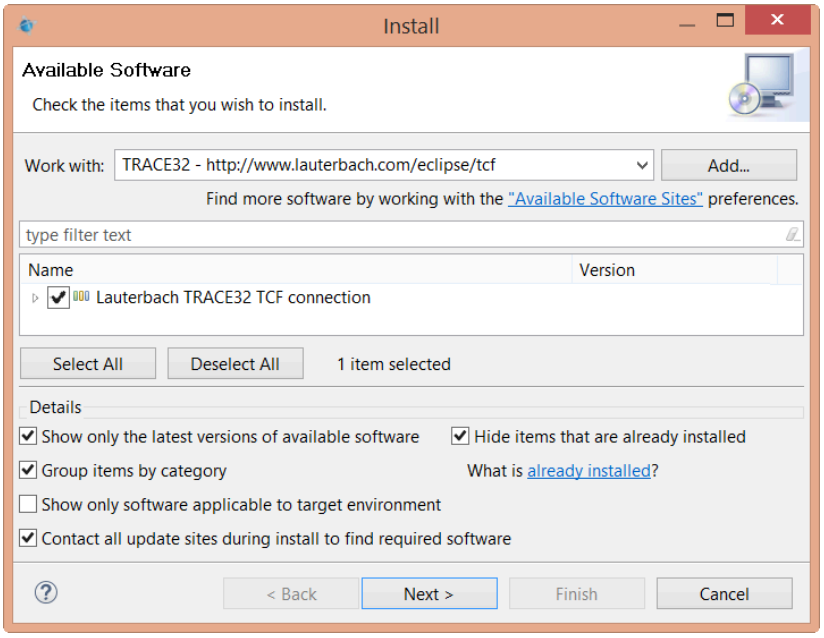
QNX Tool Suite | Integrated Development Environment User's Guide | Developer | Setup

Lauterbach offers an Eclipse plugin that lets you configure the TRACE32-ICD through a debug configuration created in the IDE. This is simpler than using a configuration file or the **T32Start** program.

For details about the plugin, see the *TRACE32 as TCF Agent* Lauterbach document (**app_tcf_setup.pdf**) found on the TRACE32 download page. Here, we explain the steps for installing the plugin:

1. In the IDE, select **Help** > (and then)**Install New Software**.
2. Click the **Add** button in the upper right corner of the **Install** window, next to **Work with**.
3. In the **Name** field of the **Add Repository** window, enter: TRACE32
4. In **Location**, enter: `http://www.lauterbach.com/eclipse/tcf`
5. Click **OK** to close the popup window.

   The list area of the **Install** window now shows entries for the software available from this location.
6. Check the box next to `Lauterbach TRACE32 TCF connection`.



7. Click **Next** twice, to advance to the **Review Licenses** dialog.
8. Accept the license agreement, then click **Finish**.

   The window closes and the IDE begins downloading the Eclipse plugin. Progress is displayed in the bottom right corner. You may see another window asking if you trust the certificates of the software packages. If you see this, check the box next to the Lauterbach list entry and click **OK**.
9. When the download finishes and the IDE prompts you to restart Momentics, click **Yes** to complete the installation.

   When the IDE reopens, you'll see a new menu, **TRACE32**. This menu lets you run and edit startup scripts, which are explained later, and attach and detach the debugger to and from the target.

Page updated: August 11, 2025

JTAG: Debugging with Lauterbach TRACE32

QNX Tool Suite | Integrated Development Environment User's Guide | Developer | Setup

The Lauterbach TRACE32® In-Circuit Debugger (TRACE32-ICD) supports the JTAG onchip interface so you can perform hardware debugging of QNX OS systems through the QNX Momentics IDE.

This debugger communicates with the IDE through the Target Communication Framework (TCF). This framework supports full-featured debugging, using the TRACE32-ICD as the debugging engine.

The sections in this appendix explain how to install, configure, and use the TRACE32-ICD with a QNX OS image. The tasks involved are:

- Prerequisite setup
- Installing the TRACE32 software
- Installing the TCF in the IDE
- Installing the TRACE32 TCF Eclipse plugin
- Building applications and an OS image for debugging
- Creating a TRACE32 startup script for your QNX image
- Creating a debug configuration that uses the script
- Starting a debugging session

Page updated: August 11, 2025

Prerequisites

There are hardware and software requirements for the Lauterbach TRACE32 In-Circuit Debugger.

Before you can install, configure, and use the TRACE32-ICD, you must verify that you have the following hardware and software:

- Hardware requirements:
  - Lauterbach PowerDebug or PowerTrace module
  - an appropriate JTAG debug cable for your specific target board
  - a USB or Ethernet cable to connect the debugger to your host
  - For the list of supported target architectures, see the Supported Processor Architectures heading on the left side of the Lauterbach homepage.
- Software requirements:
  - Lauterbach TRACE32 Software Installation dated February 2016 or later
  - QNX Momentics IDE version 7.0 or higher

Page updated: August 11, 2025

Starting a TRACE32-ICD debugging session

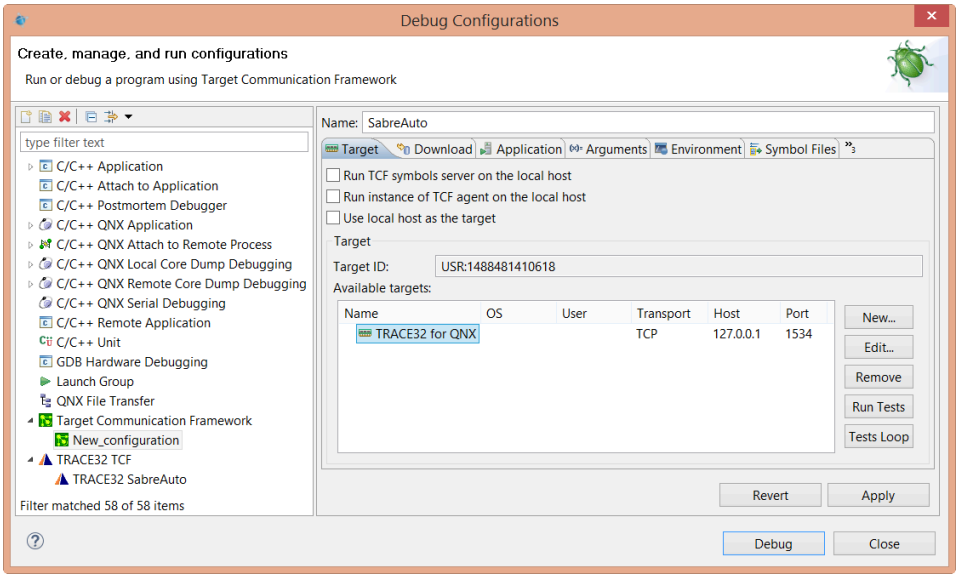
When the QNX OS image is running on your target, the TRACE32 TCF plugin is installed in the IDE, and you've created a debug configuration, you're ready to debug your target using the TRACE32-ICD.

To start a JTAG debugging session using the TRACE32-ICD:

1. Click the dropdown arrow in the TRACE32 button ( ) and select your debug configuration to start the TRACE32 software.

   The IDE adds this button near the launch bar when you install the TRACE32 TCF plugin.
2. When the TRACE32 software starts, wait until it completes execution of the startup script.
3. Select **Run** > (and then)**Debug Configurations**.
4. In the list on the left, double-click **Target Communication Framework** to create a new configuration.

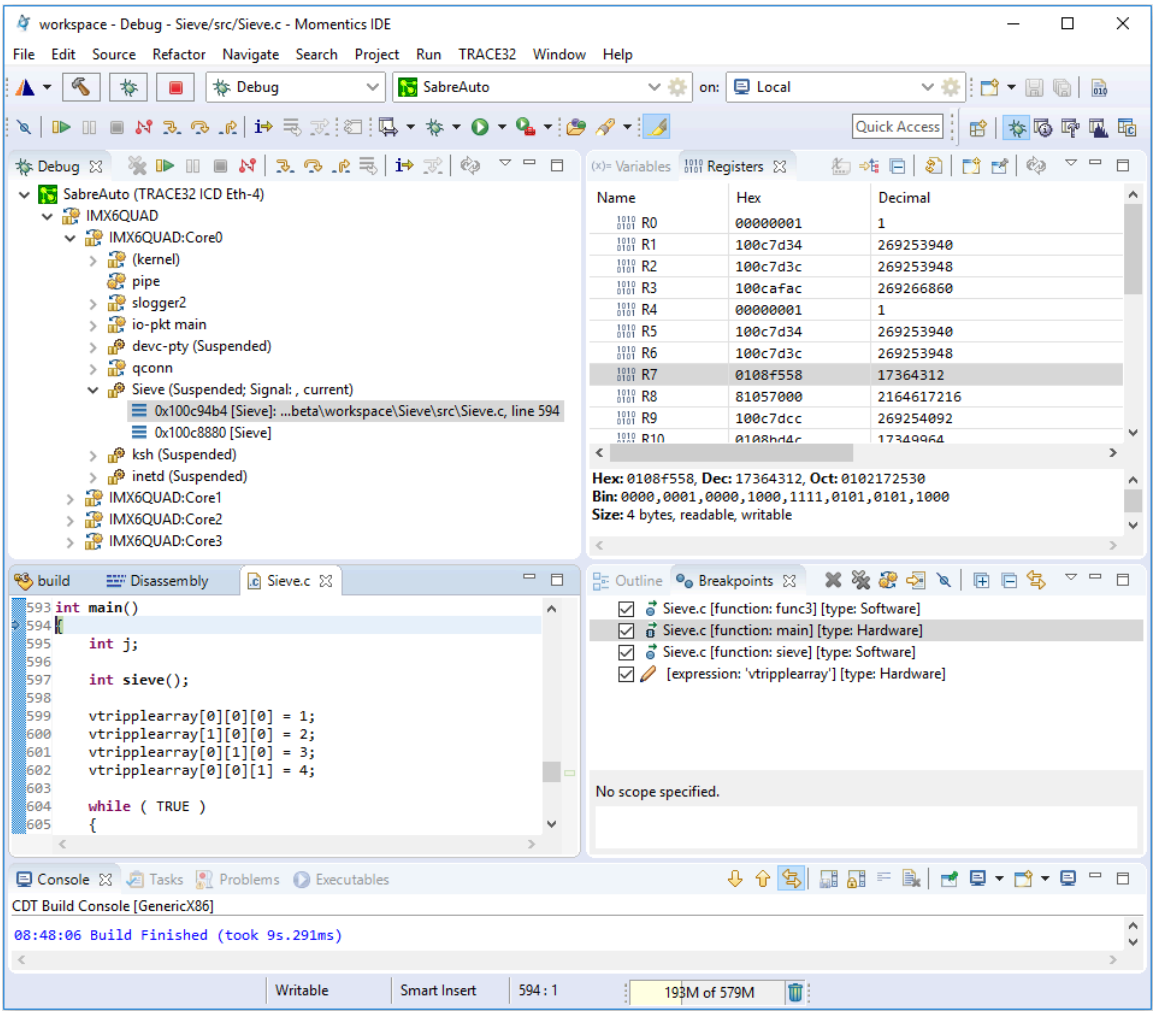
   The configuration tabs appear in the main area of the window.
5. Provide a unique name in the **Name** text field at the top.
6. Click the **Target** tab on the left to access the fields for configuring the target for the TCF connection.
7. Clear the **Use local host as the target** checkbox.
8. In the **Available targets** list area, select TRACE32.



9. Click **Debug**.

The IDE switches to the Debug perspective starts a debugging session by connecting to the TRACE32 software through the TCF plugin. The TRACE32 software controls the TRACE32-ICD module.

The Debug views display the target processes and threads, the source code, register contents, and other essential debugging information. The debugging controls let you set breakpoints, step through the code, and stop and resume execution.



Page updated: August 11, 2025

Controlling kernel event tracing through API calls

In your application code, you can use the *TraceEvent()* function to control what kernel event data are written to memory and when the data are captured to a file. Calls to this function modify how the instrumented kernel logs events.

Adapting your code to use this API (and rebuilding and relaunching the application) is admittedly more work than configuring kernel event trace settings in the IDE or specifying `tracelogger` options. However, the API allows you to turn instrumentation mode on and off, emit events for only certain classes and types, and insert custom events into the trace.

For information about all commands that control kernel event logging, see the *TraceEvent()* entry in the *C Library Reference*. Here, we demonstrate how to use some common commands.

Sample program

Your program must include the trace header file (**sys/trace.h**). *TraceEvent()* is the only function it must call but as mentioned, these calls are effective only when the kernel's instrumentation mode is active, meaning it's logging events.

In our case, we want close control of kernel instrumentation, so we run `tracelogger` in daemon mode, using the `-d1` option. This option makes the utility wait for the associated *TraceEvent()* commands before enabling kernel instrumentation or capturing data to the kernel event log (**.kev**) file. Details on daemon mode are given in the `tracelogger` entry in the *Utilities Reference*.

Consider the following program:

```
#include <stdlib.h>
#include <stdio.h>
#include <process.h>
#include <sys/trace.h>

#define NUM_PROCESSES 16

int main() {
  /*
     To keep the amount of data generated manageable, we define a
static rules filter.
     We do so by first disabling all tracing by suppressing tracing for
all classes
     and for process- and thread-specific events, which must be cleared
separately.
  */
  TraceEvent(_NTO_TRACE_DELALLCLASSES);
  TraceEvent(_NTO_TRACE_CLRCLASSPID, _NTO_TRACE_KERCALL);
  TraceEvent(_NTO_TRACE_CLRCLASSPID, _NTO_TRACE_THREAD);
  TraceEvent(_NTO_TRACE_CLRCLASSPID, _NTO_TRACE_SYSTEM);
  TraceEvent(_NTO_TRACE_CLRCLASSPID, _NTO_TRACE_COMM);

  /*
     Next, we tell the kernel to emit only certain event types and to
use wide mode,
     meaning it outputs more event data. With our sorting algorithm,
the processes
     send each other data using many-to-many exchanging. So, we want
```

Based on the *TraceEvent()* calls, every process in this program emits a custom event when it begins a new phase in sorting. These events make the kernel event trace easier to navigate, by allowing you to quickly find when a given process began a particular phase, within the timeline of all events.

Aside from the custom events, our program emits only:

- thread events for entry into the Ready, Running, Send, Receive, and Reply states
- communication events for message Send, Receive, Reply, and Error calls

If you want even more control over kernel event logging, you can manage the kernel buffers by issuing commands to allocate memory for them, specify the mode for writing kernel events into them, and flush their contents when needed. These advanced steps are explained in "Using *TraceEvent()* to control tracing" in the *System Analysis*

Toolkit User's Guide.

Page updated: August 11, 2025

Interpreting trace data outside of the IDE

The System Profiler provides a flexible and powerful tool for viewing kernel event trace data. But if you prefer to directly read and interpret kernel event log (**.kev**) files, you can use non-IDE tools.

There are two ways to analyze kernel event trace data outside of the System Profiler:

`traceprinter`
: This command-line utility parses the events in a log file and sends their data to standard output. The tool is designed for displaying text descriptions of kernel events captured by `tracelogger`.

  For full details, see the `traceprinter` entry of the *Utilities Reference*.

traceparser*()
: This set of C library functions lets you interpret logged kernel data within your own programs. It defines a parser for reading the specified log file and supports creating callbacks to process specific events.

  For details on this API, see the *traceparser()* entry of the *C Library Reference*.

Page updated: August 11, 2025

Kernel event trace log configuration

The IDE uses log configurations to remember settings for running kernel event traces. You can select a log configuration and perform a kernel event trace from the launch bar or the **Target Navigator**. The log configuration fields let you filter what information gets logged and set how long the trace runs.

The preferred way of accessing a log configuration is through the launch bar. The configuration editor that gets opened displays multiple tabs for entering the configuration fields. The fields in each tab are described in the tables that follow. The **OK** button at the bottom of the editor saves the current settings and closes the window, while **Cancel** closes it without saving the settings.

If you go through the **Target Navigator** and access the **Log Configurations** window, you'll see the different log types in the left pane, with an entry (**Neutrino Kernel Event Trace**) that lists the available kernel event trace configurations. When you select one of them or create a new one (by clicking that same entry and then **New launch configuration** (📄) in the upper left corner), the window displays the same tabs but also has a text field at the top for entering the configuration name.

Table 1. **Main** tab fields

Name	Description
Save Log in	The location to save the log file containing the kernel event trace data. You can: • leave the field blank to use the default location, which is the project directory for the target • enter an absolute path (e.g., **C:\Users*username***) • click **Browse Workspace...** to select a workspace location from a project explorer • click **Browse Filesystem...** to select a local directory from a file selector
Save Log Configuration as	Whether to save the configuration locally, so it's accessible to only those users with access to the chosen directory, or in a shared location, so it can be shared with developers working on other projects. For the shared setting, you must either enter a project location (e.g., ***project_name*\logs**) or click **Browse** to select a location from the project explorer.

Table 2. **Trace Settings** tab fields

Panel	Name	Description
Tracing duration	**Period of time**	This radio button lets you limit the data capture by time. The value that you enter in the adjacent text field sets the number of seconds that the kernel event trace runs for. **NOTE:** Any kernel event trace should be limited to a few seconds because longer traces produce too much data to be useful.
Tracing duration	**Number of iterations**	This radio button lets you limit the amount of data captured. The text field must contain a positive integer to set how many kernel event buffers are used.
Trace collection	**Save on target then upload**	Click this radio button to make `qconn` write the kernel event buffers to a file on the target and then upload that file to your workspace.
Trace collection	**File path on target**	The target location for writing the kernel event log file. In the text field, you can enter an absolute path that includes the filename or a relative path. In the latter case, the file is written to **/dev/shmem/**.
Trace collection	**Memory mapped file then upload**	Click this radio button to make `qconn` write the kernel event buffers to a memory-mapped file on the target and then upload that file to your workspace. With a memory-mapped file, you can limit the amount of kernel data logged, based on file size.

Panel	Name	Description
Trace collection	**Filename on target**	The name of the kernel event log file. This file is written to **/dev/shmem/** because it must be kept in shared memory. In the text field, you can enter only a filename and not a path.
Trace collection	**Max file size**	The maximum size for the log file. If the file reaches this size, any additional data are discarded. Standard byte size modifiers—B, K, M, and G—are allowed. If no modifier is given, the bytes unit is assumed.
Buffers	**Number of kernel buffers**	The size of the static ring of buffers allocated in the kernel.
Buffers	**Number of qconn buffers**	The maximum size of the dynamic ring of buffers allocated by `qconn`.

Table 3. **Event Filters** tab fields

Name and scope	Description
Mode (General)	A general setting for the amount of information to capture in the log file. One of: • `Fast` – Less information is captured, which means less work for the kernel and a smaller file. • `Wide` – More information is captured, which means more work for the kernel and a larger file. • `Class Specific` – The amount of information captured varies with the event class.  **NOTE:** The other **Event Filters** fields aren't shown if you select `Fast` or `Wide`. If you select `Class Specific`, the UI displays fields that you let set the mode for each class.
Mode (Class-specific)	The amount of information to capture for events of a particular class. This field is shown for: • Kernel Calls • Interrupts • Process and Thread • System • Communication • Security This field can be set to `Disable` (to disable logging for a given class), `Fast` or `Wide` (which have the same meanings as they do at the general level), or `Event Specific` (to configure logging for individual event types). With this last setting, the UI displays a grid with a mode selector for each event type. The default mode is `Fast` for all event classes, except for Process and Thread, for which it's `Event Specific`.
Name (Event-specific)	The name of the event type. Click the **Mode** entry in a row to change the logging mode only for that event type. Multi-select is also supported, by holding the **Ctrl** and/or **Shift** keys when selecting event entries. After you select multiple event types, you can change the logging mode for all of them by clicking the **Set All Selected Properties To...** button at the bottom, then selecting the mode from the dropdown list in the **Properties Editor** dialog box.

Name and scope	Description
Mode (Event-specific or in **Properties Editor**)	The amount of information to capture for the selected event types. This setting can be `Disable`, `Fast`, or `Wide` (which have the same meanings as they do at the class-specific level). For the Process and Thread event class, the default mode is `Fast` for all event types, except for Running, for which it's `Wide`. For all other event classes, all individual event types will be initially set to `Fast` mode when you select a class-specific mode of `Event Specific`.  **NOTE:** For information about each trace event, including the data emitted for the `Fast` and `Wide` modes, see the Current Trace Events and Data chapter in the *System Analysis Toolkit User's Guide*.

Table 4. **Address Translation** tab fields

Name	Description
Enable address translation	A checkbox that turns address translation on or off. Address translation makes the IDE map each function event to a source file and line number and display these data in the **Trace Event Log** view. In the Timeline pane in the System Profiler results, you can show data related to address translation.
Binary Locations	A tab that lists the search locations for binaries. The log file content must be matched with the binary files in your workspace for address translation to occur. The **Add** button on the right brings up the project explorer, allowing you to select a project folder to search for binaries in. Other buttons let you remove locations or change their search order.
Binary Mappings	A tab that provides a grid listing of all binaries on which address translation can be performed, and their load addresses and process names. You must first add search locations in **Binary Locations** before you can define mappings (i.e., translations). The **Add Binary** button on the right opens the **Binary Object Selection** window, where you can enter the filename of an executable or click **Browse** to select an entry from the list of executables found in the previously specified locations. The same window also lets you override the executable binary's default load address and process name. To perform address translation on a library used by an executable binary, select the binary's entry and click **Add Library**, which also brings up the **Binary Object Selection** window. Here, you must define the shared object name, and you can override the library's default load address. For information about disabling default address loading, see the next subsection. In **Binary Mappings**, you can click **Edit** to redefine the binary name, load address, or process name for the selected entry. There's also a **Remove** button to delete it. Finally, the **Import** button opens a file selector that lets you pick a file that contains the output of the `pidin mem` command. When you pick such a file, the IDE populates the list with information about all processes listed in the `pidin` results that were redirected to the file. Note that you must first run such a command on the target (e.g., `pidin mem > pidin_results.txt`). In order to prevent truncation of binary names, the usage of the following `pidin` command is recommended: `pidin '-F%a %b %60N %40h %p %J %c %d %m'  '-M %80M @%> %? %< %='`

Address Translation

Address translation is an IDE feature that maps the virtual address of each function listed in the kernel event trace log file to a source file and line number. By revealing function locations, the translation makes it easier to interpret events; when it's disabled, only virtual addresses are shown.

You can configure this feature through the **Address Translation** tab. By default, the **Binary Object Selection** window opened by the **Add Binary** and **Add Library** buttons specifies that the load address of the binary file should be automatically discovered. This makes the IDE analyze the log file to find library addresses, which it then displays for function events. Any discovered addresses are stored in the **Trace Event Log** data so the lookup isn't done every time you open the log file.

NOTE:
Address translation is automatically done for any applications built with function instrumentation and run during a kernel event trace.

If auto-discovery of library addresses is unsuccessful, the IDE warns that you need to manually set the address information. You can disable the warning by selecting **Window** > (and then)**Preferences** > (and then)**QNX** > (and then)**System Profiler** > (and then)**Address Translation Configuration**, then unchecking **Provide a warning if auto-discovery address translation fails**. But if want to see the addresses, you should check the **Event Filters** tab to ensure that for System events, the generation of MMap Name and MMap Name (64) events is enabled.

If you uncheck the **Auto-discover load address** box, you must enter a nondefault address in the text field above. You would only do this if there was a bug that you needed to work around.

Page updated: August 11, 2025

Controlling kernel event tracing

The IDE lets you easily configure and start kernel event tracing. But sometimes, you might not have a target connection for controlling tracing. At these times, you can use the `tracelogger` utility to capture trace data. To further control tracing, your applications can call the kernel event trace API.

Initiating kernel event traces

When you run a kernel event trace from the launch bar, the IDE tells `qconn` of the initial trace settings and to start capturing event data. The trace settings are based on the UI fields in the log configuration.

You can also run the `tracelogger` command-line tool on the target. The advantage of this approach is that you can start a trace automatically (i.e., using a script) at a specific operational phase, including during system startup when you can't use the IDE because there's no active target connection. In this case, the trace settings are based on the options given to `tracelogger`.

Controlling kernel event tracing at runtime

Unlike **librcheck** or **libprofilingS**, `qconn` and `tracelogger` don't accept signals for requesting common operations such as starting and stopping tracing. For `tracelogger`, there are a couple of exceptions:

- If you start it in ring mode using the `-r` option, the service doesn't capture events until you send it a SIGINT signal, at which point it flushes the kernel buffers to the log file.
- If you run it without specifying a data-capture limit, either with `-s` to set a time limit or `-n` to specify a fixed number of buffers to output, the service runs until you explicitly terminate it using **CtrlC** or the SIGKILL signal.

Both `qconn` and `tracelogger` use *TraceEvent()* to control the kernel's event-gathering module. Your applications can make *TraceEvent()* calls to closely control what event information is captured. For these calls to have any effect, though, the kernel's instrumentation mode must be active.

The best technique is to use the IDE or command line to start a trace before or after you begin running applications that use *TraceEvent()*. This way, `qconn` or `tracelogger` does the tedious work of managing buffers, collecting trace data, and saving the data in the appropriate form, while allowing the applications to choose which events get logged and to start and stop data capturing. Any adjustments made at runtime override any equivalent initial settings defined when the trace was launched.

Page updated: August 11, 2025

How kernel event tracing works

Kernel tracing is an analysis activity in which the instrumented kernel on a target logs information about kernel events. This information shows what's happening on the target at a system-wide level, so you can eliminate performance bottlenecks and optimize the interaction of processes and threads.

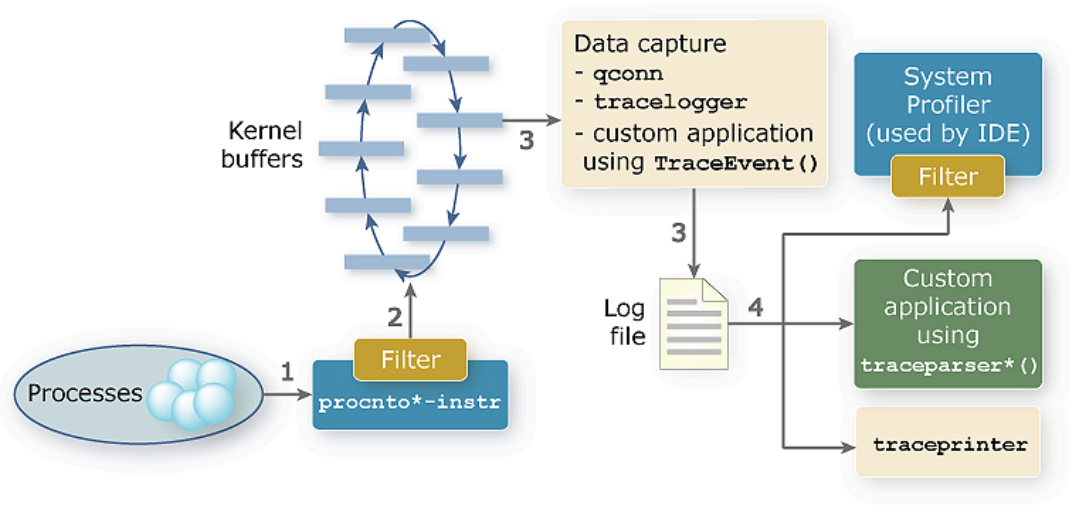
In the IDE, the System Profiler component allows you to examine the data generated by kernel event traces. These data describe events such as:

- kernel calls
- interrupts
- process and thread management activities
- system events related to adaptive partitioning and memory mapping
- context switches within a process
- message passing and signaling

Kernel event trace: procedure overview

Several components work together to generate, capture, and present kernel event data. You must run the instrumented QNX OS kernel and a data capture program on the target to collect and output the trace data, and run a client program (e.g., the System Profiler) on the host to read the data:

Figure 1Kernel event trace procedure: information flow



Kernel event tracing works as follows:

1. **Generating event information**

   The instrumented kernel (**procnto*-instr**) contains an event-gathering module. This module continuously generates information about the system-level activity (e.g., kernel calls or context switches) of active processes and threads. Program code can also generate such information by using the *TraceEvent()* kernel call.

2. **Logging of events**

   When requested, the instrumented kernel writes time-stamped and CPU-stamped events to a set of buffers. The default number of kernel buffers is 32, but you can set a different number in the **Trace Settings** tab in the **Log Configurations** window. Although the number of buffers is limited only by the available system memory, it's important that this space is managed carefully. If all types of events are traced, the amount of information emitted can be quite large.

   You can filter the kernel events that get written, by adjusting the **Event Filters** fields in the log configuration. When you view trace data, you may find only some information to be relevant and decide to restrict which events get logged by the kernel. Filtering lets you run longer traces that still produce a manageable amount of data, improves system performance during tracing, and makes the trace results more readable.

   When you start a trace, the IDE tells `qconn`, which calls *TraceEvent()* to tell the kernel to start writing event data to the buffers and to inform it about all trace preferences, including filters.

3. **Capturing data**

The `qconn` process captures event data by transferring it from the kernel buffers to the output specified in the configuration. The output is either the System Profiler (if streaming output is requested) or a local file on the target (if a target path is given).

Data capturing is done for the duration that's specified along with the output location and number of `qconn` buffers in the **Trace Settings** tab. The `qconn` buffers temporarily store data after reading them from the kernel buffers but before writing them to the output. You can change the number of `qconn` buffers from its default of 128, to find the right balance between the memory used and the amount of information that the process can hold.

NOTE:
There's nothing in `qconn` that prevents data loss if it can't keep up with the kernel's writing of data.

You can capture data with other tools such as the `tracelogger` utility found in QNX SDP. Like `qconn`, this other utility uses *TraceEvent()* to filter kernel events. You can also write a custom data-capturing application that uses this same API function, as explained in "Using *TraceEvent()* to control tracing" in the *System Analysis Toolkit User's Guide*.

4. **Analyzing data**

   The System Profiler provides the best way of analyzing large amounts of kernel event trace data. When the trace finishes and all of its data have been streamed to the IDE or stored in a kernel event log (**.kev**) file and uploaded to the host machine, the IDE saves the data locally.

   When you open a trace log file in the IDE, the editor pane displays graphical charts summarizing the captured trace data. You can change which trace statistics are shown by switching panes in the editor, and define filters to display a subset of the trace data.

   In addition to the IDE, you can use the `traceprinter` utility (which is part of QNX SDP) to analyze trace data. This utility parses and prints the data in **.kev** files written by `tracelogger`. Also, you can write a custom application that uses the *traceparser*()* system calls to parse trace data so you can then print them in a convenient format, as explained in "Building your own parser" in the *System Analysis Toolkit User's Guide*.

Page updated: August 11, 2025

QNX launch configuration properties

The launch configuration properties are defined in multiple tabs and fields that vary with the launch configuration type. For those types that support QNX projects, you can configure C/C++ project properties, runtime analysis tools, and settings for uploading binaries to a target.

NOTE:
You can find information about the Eclipse launch configuration templates in the **Tasks** > (and then)**Running and debugging projects** > (and then)**Creating or editing a run/debug configuration** entry of the *C/C++ Development User Guide*.

Also, the properties for Neutrino Kernel Event Trace are completely different than those of other QNX launch configuration types. The QNX File Transfer configuration type shows only the **Common** tab and its own **File Transfer** tab.

Main tab

Identifies the project, binary, build policy, and launch target. This tab is displayed by all launch configuration types designed for running, debugging, or analyzing an application.

Project

The project that this configuration applies to. You can enter the project name in the text field or click **Browse** to select the project from the workspace.

C/C++ Application

This field tells the IDE which binary file to run. You can name a file by specifying an absolute path on your host or a relative path within the project.

For some configuration types, there's an **Auto-pick** checkbox, which indicates whether the IDE should automatically select the binary file. By default, this box is checked. Automatic selection is convenient but it takes longer because the IDE must look for the binary file.

If you fill in the **C/C++ Application** text field and leave the box checked, the IDE treats the text field value as a hint. For example, suppose you have a project with two binaries—*alpha* and *beta*—and you compile the project for the x86_64 and AArch64le architectures. If the text field says **build/x86_64-debug/*alpha*** but you launch the project on an AArch64 target, the IDE auto-picks **build/aarch64le-debug/*alpha*** (and not **build/aarch64le-debug/*beta***).

NOTE:
If you leave the text field blank, the IDE always auto-selects the binary file, even if **Auto-pick** is unchecked. Checking this box makes a difference only when you've named a binary file.

You can also specify an IDE variable as the text field value, by clicking the **Variables** button and selecting from the resulting list. Also, the **Search Project** button lets you pick from the list of binary files found within the project, while the **Browse** button brings up a file selector so you can pick a file from the local filesystem.

Build Configuration

The build configuration to use. This selection determines which binary version gets built (if required) before the project is launched. Note that this setting is significant only when **Auto-build** (explained below) is enabled.

The **Build Configuration** dropdown lists the build configurations for specific architectures and use cases (e.g., `x86_64-coverage`, `aarch64le-debug`), assuming your project uses standard makefiles. If it uses recursive makefiles, you'll see the **Configuration** option, which means the QNX C/C++ Project properties determine which binary versions get built.

For all projects, there's also the **Use Active** option, which tells the IDE to build the binary for the active build configuration, and the **Select Automatically** option, which makes the IDE select the build configuration based on the launch mode.

Auto-build

This set of radio buttons lets you choose an auto-building policy. The options are: **Enable**, **Disable**, or **Use workspace settings**.

When auto-building is enabled, if necessary, the IDE builds the appropriate binary file before running it on the target. When auto-building is disabled, if the IDE doesn't find the right binary file, an error message is

displayed if you try to launch the project. The third option is handy if you want to use the same auto-build policy for all launch configurations.

Target

The target for running an application when you launch it using one of these two methods:

- from the launch configuration manager, by clicking **Run** (or the equivalent task button) in the bottom right corner
- from the **Project Explorer**, by right-clicking a project and selecting **Run As**, **Debug As**, or **Profile As**, then the launch configuration type from the submenu listings or the configurations selector, and then the launch configuration itself from the popup list

This property also tells the IDE which target connection to use to populate the file navigators for other properties, such as the remote directory for running the binary file. You can enter the target name into the text field or click **Browse** to select from the list of launch targets.

Arguments tab

Lists the command-line arguments, working directory, and some properties of the binary that gets run. All launch configuration types that start a process display this tab, for all launch modes except Attach.

C/C++ Program Arguments

The command-line arguments to pass to the executable binary. You don't need to include the executable file's name and you must separate the arguments with spaces (not commas or other characters). For example, if you want to send the equivalent command line of `HelloWorld -v -L 2`, type `-v -L 2` into this text field.

The `${string_prompt}` value instructs the IDE to prompt you for an argument every time you launch the application. You can have multiple entries like this, to display a prompt window for each input argument. Also, you can provide each prompt with its own label, by specifying values in the form of `${string_prompt:`*`prompt_text`*`}`, where *`prompt_text`* is a text string without quotes and possibly containing spaces.

Working Directory

The target directory in which the executable runs. Initially, this box is unchecked, which means the executable runs in the same directory that it's uploaded to. This upload directory is defined by the **Remote directory** field in the **Upload** tab. If you uncheck the box, you can enter a different directory path in the text field or select a directory from a file selector.

Adaptive Partition

When you check this box, you can select the adaptive partition in which to run the executable. For an explanation of the concept and benefits of adaptive partitioning, see the Adaptive Partitioning chapter of the *System Architecture* guide for the OS version that your target is based on.

Priority and Scheduling

Checking this box lets you set the priority and scheduling algorithm for the threads of the application process. Information on thread priority and scheduling is found in the "Thread Scheduling" section of the *System Architecture* guide.

Test Runner

Checking this box lets you select a test framework to run unit tests. For information about the supported test frameworks and running test programs, see the Unit Testing chapter.

Environment tab

Defines environment variables for the target. This tab is displayed by all launch configuration types that support starting a process, for all launch modes except Attach. No variables are set by default, but you can add variable definitions with the following buttons:

- **New** – Opens a prompt to enter a variable name and value. For the value, you can select an IDE variable (e.g., *string_prompt*).
- **Select** – Opens a window that lists the environment variable settings of the host, allowing you to add any subset of those variable settings to the launch configuration.
- **Import** – Opens a file selector that lets you choose a local file to read to populate the list of environment variable settings. The selected file must be a text file, with one name-value pair per line, in this format: *`VARNAME=value`*

There are also buttons on the side to edit or remove existing variable definitions, and radio buttons at the bottom to choose whether the environment variable settings shown here should be appended to or override the settings on the target.

Upload tab

Controls how the executable binary and other files are uploaded from the host to the target. All launch configuration types that involve starting a process display this tab, for all launch modes except Attach.

Upload executable to target

This option tells the IDE to upload the executable binary to the target (if necessary) when launching the project. When this radio button is selected (as by default), the text field is called **Remote directory** and it stores the location that the binary is uploaded to. You can either enter an absolute path or click **Browse** and select a directory from the file navigator. This directory must contain any shared libraries needed by the application, or you must upload those libraries by using the **Upload shared libraries** setting.

There are two checkboxes that also appear when this radio button is selected:

- **Strip debug information before uploading** — When checked, the IDE removes debug information from the binary file before uploading it. This makes the file smaller and faster to upload, but means that you need a copy of it on the host to debug your program.
- **Use unique name** — When checked, the IDE appends a number to the binary's filename to make it unique each time it's executed.

Use executable on target

This option tells the IDE to run an executable binary found on the target. When this radio button is selected, the text field is called **Remote executable** and stores the location of the executable. You can either enter an absolute path or click **Browse** and select a directory from the file navigator. This directory must contain any shared libraries needed by the application, or you must upload those libraries by using the **Upload shared libraries** setting.

Upload shared libraries or other files to the target before launch

When this box is checked (as by default), the IDE uploads the listed files to the target when launching the project. These files can be libraries, executables, or data files.

The **Auto** button populates the list with all libraries missing on the target. A library is "missing" if it's stored in a user (not system) path and listed in the NEEDED section of the executable binary. The **Add** button opens a window that provides a pre-populated list of workspace files but also lets you choose a file from the workspace or local filesystem.

Each entry shows the file's name, its host and target paths, and these two checkboxes:

- Upload — indicates whether this file is included in the set of files uploaded
- Strip — indicates whether to remove debug information before uploading the file

If you uncheck the **Upload shared libraries** box, the list is greyed out and no files are uploaded. However, the entries aren't deleted, meaning you can later reenable the uploading.

Remove uploaded components after session

Makes the IDE remove, from the target machine, the executable binary and any other uploaded files after the application finishes executing.

Debug tab

Enables or disables debugging and defines debugger options. The IDE uses the GNU Debugger, GDB. All launch configuration types that support debugging display this tab.

Debugger

This checkbox turns debugging on or off. When it's checked, you can see and set the debugger options. When the box is unchecked, the IDE hides the debugger options and doesn't attach the debugger when you launch the application.

Stop on startup at

When this option is set, the debugger stops when it enters the function named in the text field (by default, *main()*). The text field must be filled in when the box is checked.

GDB debugger

The GDB version to use. The default value of `gdb` means the IDE selects the version based on the architecture of the executable. The IDE comes with several GDB versions, each of which is named according to its supported architecture (e.g., **ntoaarch64-gdb**). Information on the debugger is given in the `gdb` entry in the *Utilities Reference*.

The only time you might want to change this field is if you're running a different `gdb` binary (e.g., a patched version) or if the IDE fails to pick the correct debugger version.

GDB command file

Provides the path of a file containing commands to be executed by GDB. The default path is **.gdbinit**.

Load shared libraries automatically

When this option is set, the debugger loads shared libraries used by the application and found on the host, so you can step through their code. You can uncheck this box to save memory on the host, because library symbols can consume significant memory.

Source tab

Defines the locations in which the debugger should look for source files. This tab is displayed by launch configuration types that support debugging.

NOTE:
You need to configure these settings only when debugging code not compiled on the host. For host-compiled code, the debugger can find the source files without hints from the user.

Source Lookup Paths

Lists the "source containers", which define the paths for finding source code when debugging. The search order of the paths is from top to bottom in the list.

Initially, the list shows only one item, entitled Default, that groups three containers. You can't modify its contents but you can remove it and define your own containers.

If you click the **Add** button, the **Add Source** window appears and lists these container types:

Absolute File Path

Use any source file path compiled into the binary. The IDE interprets all paths "as is", meaning if you're debugging a source file compiled with a path of **/foo** on another machine, your host machine must store that file at the same absolute path for the IDE to find it when debugging.

Compilation Directory

A local directory used for running the C/C++ compiler. You can specify the directory in the popup selector, which lets you browse the filesystem.

File System Directory

A local directory. This container type is similar to Compilation Directory except you can name any local directory, not just one used by the C/C++ compiler.

Path Mapping

A mapping of paths compiled into the binary to equivalent local paths. For instance, if the compilation path for some source files is **/foo** but the host location of these same files is **/tmp/foo**, you must define a mapping between these directories to allow the debugger to find the files on the host.

Program Relative File Path

Use any relative path compiled into the binary. You should use this container type when your host's directory structure matches that of the development system.

Project

One of more workspace projects. You can select these projects from a file selector. For each container of this type, the IDE searches recursively within the project, without considering the relative paths of the source files.

Project - Path Relative to Source Folders

This container type is similar to Project except that the IDE examines the relative paths in the compiled-in source file locations.

Workspace

All projects in the workspace.

Workspace Folder

One or more workspace folders. You can select these folders from a file selector.

If you need to revert to the original lookup paths, you can click the **Restore Default** button. There are also buttons for editing the location in directory-specific containers, deleting containers, and moving them up or down in the list to change the directory search order.

Search for duplicate source files on the path

When checked, the IDE continues to search the lookup paths for a source file with the same name even after finding the first file with a matching name.

Common tab

Specifies how the launch configuration is stored, displayed in toolbar menus, and other properties such as the standard input and output to use. All launch configuration types display this tab, but for QNX File Transfer

configurations, only the **Save As** and **Display in favorites menu** fields are visible.

Save as

When you create a launch configuration, the IDE saves it as a **.launch** file. You can save this file in one of two locations:

- **Local file** – The IDE stores the configuration file in ***workspace_dir*/.metadata/.plugins/org.eclipse.debug.core/.launches**.
- **Shared file** – The IDE stores the configuration file in the workspace. You must either name a workspace path in the text field or choose a location from the folder selector.

Display in favorites menu

Adds the configuration to the favorites lists in the toolbar menus shown on the right of the launch bar. For example, if you check the boxes for Debug and Profile, you'll see the launch configuration in the favorites list in each of the Debug and Profile toolbar menus.

Encoding

The encoding scheme to use for console output. There are radio buttons to select either the default encoding scheme of UTF-8 or another scheme from the adjacent dropdown.

Allocate console (necessary for input)

When checked, the IDE assigns a console view to receive the input.

Input File

Names a file to redirect the standard input from. In the text field, you can enter an absolute path from the local filesystem or a relative path within the workspace. The buttons below let you select a file using a file selector, or use an IDE variable to define an input file.

Output File

Names a file to redirect the standard output to. In the text field, you can enter an absolute path from the local filesystem or a relative path within the workspace. The buttons below let you select a file using a file selector, or use an IDE variable to define the output file.

Append

When this box is checked, the IDE appends (and doesn't overwrite) the output file.

Launch in background

When this property is set (as is the default), the IDE carries out the launch configuration task in the background.

Libraries tab

Defines the paths in which the IDE tools can look to find shared libraries on the host. This tab is displayed by launch configuration types that support runtime analysis tasks, including debugging.

Shared Libraries Path

Lists the paths for finding shared libraries. The list entries can name files or directories, in the workspace or local filesystem. The search order of the paths is from top to bottom.

When you fill in the **Project** field in the **Main** tab, the IDE adds a list entry entitled System Paths. This entry contains the paths of any library files in the standard C or C++ library that are used by the project. You can't remove this entry but you can define your own entries. Also, you can uncheck any entry to exclude a file from the search paths.

The list also contains the Uploading Paths entry, which is auto-populated with the paths of any shared libraries named in the **Upload** tab.

The **Auto** button populates the list with all of the user libraries found on the host and listed in the NEEDED section of the executable binary. To add a directory to the list, click **Add Path**. This button brings up a window that lets you enter a folder path or pick a folder using a file selector.

To add a library file to the list, click **Add Library**. This button brings up a similar window, except that you're restricted to selecting a file.

There are also buttons for editing the path named in an entry, removing entries, and moving them up or down in the list, to change the search order of the library paths.

Validate and auto-populate library paths automatically prior to launching

When this box is checked (as by default), the IDE verifies that the library paths are up-to-date and if necessary, redefines those paths before each launch. You can uncheck this box after your target setup is done and you know that the library paths won't change anymore.

Tools tab

This last tab allows you to configure runtime analysis tools, and is shown by the C/C++ QNX Application and C/C++ QNX Attach to Remote Process launch configuration types. For the Debug, Run, and Attach modes, which support multiple tools, the tab is simply called **Tools**. For the modes dedicated to specific analysis tasks—Memory, Check, Profile, and Coverage—the tab name matches the mode.

The following radio buttons let you enable and configure the tools available for the current launch mode and launch configuration type (tools that are unavailable have their buttons disabled):

- **No Tool**—lets you choose to not use a runtime analysis tool
- **Valgrind**—enables controls for selecting a Valgrind tool and configuring general and tool-specific options
- **Application Profiler**—lets you set the profiling method, scope, and function profiling options
- **Memory Analysis**—lets you specify the types of memory errors to check for, stack tracing parameters, when to take memory snapshots, and other settings
- **Code Coverage**—lets you specify the source files to analyze for code coverage, the polling interval, and more

There's also a checkbox for enabling the **System Profiler**, which can run concurrently with other tools.
Explanations of how to configure and use these tools are given in the "Integrated tools" section.

Page updated: August 11, 2025

Launch modes

Launch modes reflect general use cases; for instance, whether you want to debug a program, measure its code coverage, or simply run it. Changing the launch mode allows you to perform different tasks on the currently selected project.

NOTE:
Your selection in the **Initial Launch Mode** dialog must reflect whether your new launch configuration will be used to launch a project containing an application, library, or system image (which we call a *development project*), or to automate a task related to application setup, profiling, or debugging.

Launch modes for development projects

The following launch modes support development projects:

Run
Launch an application on a target. No IDE tool is run to profile the application process and no debugger is attached to it.

Debug
Launch an application with the debugger attached. The debugger used is the one named in the **Debug** tab of the launch configuration. For details on this workflow, see "Launching an application with the debugger attached".

Coverage
Use the Code Coverage tool to find areas of code not exercised (covered) by test cases. This activity helps you improve test case coverage to ensure no hidden bugs remain in your code.

Memory
Measure memory usage with the Memory Analysis, Application Profiler, or any of the Memcheck, Helgrind, or Cachegrind tools from Valgrind.

Profile
Profile an application using the Application Profiler or any of the supported Valgrind tools.

Check
Perform runtime checking for memory errors with Valgrind Memcheck, any of the other supported Valgrind tools, or Memory Analysis.

Attach
Attach an IDE tool to a running process on the target, to debug a program or obtain realtime profiling data. You can attach any of the Application Profiler, GDB, and Memory Analysis tools.

When building a development project, the IDE generates output files that support the selected launch mode. For instance, when Code Coverage mode is selected, the IDE builds an instrumented binary along with the data files required by the code coverage tooling.

When running an application, the launch mode determines which binary version is uploaded to the target and which perspective is opened. For Code Coverage mode, the IDE uploads the binary built to output code coverage metrics, starts running this binary, and switches to the QNX Analysis perspective. The perspective switch ensures that you immediately have access to the features that support your current task.

Launch mode for debugging over a serial link

If you're using a serial connection between the host and target and you want to debug a program over that connection, you must select an initial launch mode of Debug, then select the **C/C++ QNX Serial Debugging** launch configuration type (in the next dialog of the wizard).

Information on setting up a serial connection is given in "Serial communication".

Launch mode for kernel event tracing

The Log launch mode lets you define a launch configuration for performing a kernel event trace. This activity doesn't involve running an application but instead gathering event data from the instrumented kernel.

If you select a launch configuration designed for kernel event tracing, only the Log mode is available in the Launch Mode dropdown. There is no build activity for this mode; nothing happens if you click the Build button in the launch bar. Clicking the Log button starts the kernel event trace on the selected launch target.

Page updated: August 11, 2025

Finding memory leaks

Many IDE tools track individual heap blocks so they can detect and report memory leaks. The reported details often include stack traces of allocation points, so you can identify pointers related to leaks.

The tools used for finding leaks are the same ones used for finding memory corruption, but these two activities require distinct ways of using the tools. Often, there's a trade-off between the amount and type of leaks that a tool can find versus its setup time and overhead. More leak details typically mean more setup time and greater overhead. In general, you should try using the tools in the following order:

1. **System Information**

   In the QNX System Information perspective, you can learn which processes are using the most heap memory through the **System Resources** view. Then, you can examine the heap usage for a particular process through the **Malloc Information** view. The information shown here lets you quickly spot when the process is likely leaking memory.

   This is the best tool to use first because you can switch to this perspective at any time to see the statistics. You don't need to reconfigure, recompile, or relaunch your application, and there's no application overhead because the statistics come from the general-purpose process-level allocator.

2. **Memory Analysis**

   Memory Analysis displays memory trace data generated by the debug allocation library (**librcheck**). These data describe all allocations and deallocations, and any blocks that get leaked.

   This tool has an easy setup, doesn't require recompiling the application, and imposes relatively low overhead. One drawback is that if the application crashes, **librcheck** might not report some of the current memory leaks because memory tracing might stop working at that point.

3. **Valgrind Memcheck**

   This tool intercepts all calls to *malloc()*, *free()*, *new*, and *delete* to track dynamic-memory blocks. By examining the allocation stack traces and the pointer values in a program, Memcheck can detect different kinds of leaks, such as possibly or indirectly lost blocks.

   Like Memory Analysis, Memcheck doesn't require recompiling the application. However, it imposes noticeably more memory and CPU overhead, and checks for leaks only when the application exits.

4. **Valgrind Massif**

   This tool measures heap usage and tells you which areas of a program use the most heap memory. Massif finds blocks that aren't accessed anymore but aren't freed and thus, waste space.

   Like Memcheck, Massif significantly slows the program and reports the results only at exit time. But it is useful when you want to further reduce memory usage to improve performance.

You must run a binary built with debug information so the active tool can match binary instructions with lines of code and display the symbols (i.e., backtrace) in the results. To see symbols for shared libraries, your host machine must store debug versions of these libraries. Any binary with debug information has a bug icon () next to its name in the Project Explorer. The binary selected for running when you launch a project depends on the launch configuration settings.

Page updated: August 11, 2025

Analyzing heap memory usage with **libc** allocator API

The **libc** library contains a data structure that you can read to retrieve statistics about the current heap memory usage. These statistics are more detailed than those reported by the System Information and can be read at precise code locations.

The *mallinfo()* function returns a `mallinfo` structure that contains fields that report the used, overhead, and free heap space for memory allocations.

You can attach the debugger and read these variables, which doesn't require modifying an application's code and recompiling it. However, you may prefer to write code that queries and prints memory allocation statistics, to avoid stopping and restarting the program in the debugger (but this strategy requires recompiling). The following code provides an example of how to print some memory allocation information:

```
#include <malloc.h>
struct mallinfo info = mallinfo();

...

/* Number of arenas. Useful for recognizing contention. */
printf("n_arena %lu\n", info.hblks);

/* Current total size of the arenas. */
printf("arena %lu\n", info.arena);

/* Maximum size of the arenas. Useful for sizing applications. */
printf("max arena %lu\n", info.usmblks);

/* Current total allocated space. */
printf("total alloc %lu\n", info.uordblks);

/* Current total free space. */
printf("total free %lu\n", info.fordblks);

/* Total direct mmapped memory. */
printf("total mmapped %lu\n", info.hblkhd);
```

NOTE:
You could output the heap statistics to a file or standard error instead of standard output; the above code is just for demonstrating how to read and print the data fields filled in by **libc**.

These statistics give you insight into the overhead that the memory allocator is generating and how much memory is being handed out to your application (and thus, shows up in a `pidin` memory listing as free memory) but may not be used directly. The `mallinfo` structure is defined in the *mallinfo()* entry in the *C Library Reference*, which is based off of the *mallinfo()* entry in the Linux manual page (https://man7.org/linux/man-pages/man3/mallinfo.3.html). Alternatively, you can find a short, descriptive comment next to each field for this structure in ***QNX_TARGET*/usr/include/malloc.h**.

Page updated: August 11, 2025

Controlling **librcheck** through API calls

In your application code, you can use the *mallopt()* function to control memory checking and tracing. Calls to *mallopt()* modify the parameters and behavior of **librcheck**.

Adapting your code to use this **librcheck** API (and rebuilding and relaunching the application) is admittedly more work than configuring Memory Analysis in the IDE or sending signals or commands to a target process. However, using the API lets you perform leak checks and output heap statistics at specific places in the code.

For information about all commands for controlling memory data collection, see the *mallopt()* entry in the *C Library Reference*. Here, we demonstrate how to use some common commands.

Sample program

Your program must include the debug version of the allocation library (**rcheck/malloc.h**); otherwise, the usage of any debug commands will cause a build error.

Consider the following program:

```
#include <stdlib.h>
#include <string.h>
#include <rcheck/malloc.h>
#include <sys/neutrino.h>

void foo1() {
    char* p = malloc(40); // malloc that's irrelevant for tracing
    struct _msg_info msg_rcvd;

    // Suppose we copy in data that's too large for the memory area;
    // in this case, we read in a structure that's 48 bytes into our
    // area of 40 bytes
    memcpy(p, &msg_rcvd, sizeof(struct _msg_info));

    free(p);
}

char* foo2() {
    char* p = malloc(20); // malloc that's relevant for tracing

    // Suppose we do other work in here to store data in the memory
    // area referred to by "p"; then, we return the pointer
    return p;
}

int main() {
    // If we suspect that the first function has problems with writing
    // out of bounds, we enable boundary checking
```

Based on the tracing commands, you'll see one allocation event for *foo2()* but no allocation or deallocation events for *foo1()*. You'll also see these memory errors:

- a data write to an improper heap area (for the *memcpy()* operation in *foo1()*)
- a memory leak of 20 bytes (for the memory allocated in *foo2()* but not freed)

Page updated: August 11, 2025

Controlling **librcheck**

You can specify the initial memory checking and tracing settings for the debug allocation library, **librcheck**, through Memory Analysis in the IDE or through command-line settings. You can further control memory data collection at runtime by using Memory Analysis editor controls, sending signals to the process being analyzed, and making API calls in the code.

Defining initial settings

The initial **librcheck** settings are defined through environment variables. Memory Analysis sets some of these variables when you launch an application with this tool enabled. One notable example is that Memory Analysis automatically turns on memory tracing at the start of a program. Some programs allocate a huge number of blocks initially, making the trace output unreadable. In these cases, you would want to disable memory tracing.

When launching applications outside of the IDE, you can set the environment variables ahead of time (e.g., in a setup script) or on the command line. An example of such a command line is given in the "Using the **librcheck** library" section of the *Programmer's Guide*. Details about all environment variables applicable to the debug allocation library are given in the *mallopt()* entry in the *C Library Reference*.

Adjusting memory data collection at runtime

When an application is running, there are three methods of controlling **librcheck**:

Memory Analysis

In the Memory Analysis editor, the **Settings** tab displays many of the tool's launch configuration fields as well as buttons that trigger specific operations (for active sessions only). Using this UI, you can change which memory checks get done, enable and disable tracing, and adjust the leak check interval. You'll then see the **librcheck** changes reflected in the latest analysis results.

The results let you interpret errors uncovered by memory checks, view the allocations and deallocations found during memory tracing, and examine details about memory leaks.

Signalling

The **librcheck** library can handle four signal types that map to various commands. This lets you gather memory leak data, turn tracing on and off, and execute commands from in a file.

API calls

The *mallopt()* C library function supports many commands specific to the debug version of the allocation library. You can issue these commands in problematic areas of code, to find memory leaks or enable and disable tracing.

You can still use Memory Analysis or a command line to specify the initial **librcheck** behavior, but then use one or more runtime control methods to refine the memory checking and tracing. If you disabled memory tracing at the start of a program, you could turn on tracing just inside *main()* to see all allocations and deallocations made within the application code but not the system allocator code.

In general, any memory checking or tracing settings made at runtime override any equivalent environment variable settings that were in effect when the program was launched.

Page updated: August 11, 2025

Controlling **librcheck** through signals

You can control **librcheck** by sending POSIX realtime signals to a process that has loaded this library. Using signals lets you adjust memory checking and tracing at runtime without modifying the code or relaunching the application.

NOTE:
For this IPC mechanism to work, the library must run a control thread. You can request this setup by checking the **Create control thread** box in the Advanced Settings for Memory Analysis or by setting the *MALLOC_CTHREAD* environment variable to 2 on the command line that launches the application.

The library handles four signal types that request common data collection operations. In the IDE, you can send signals to target processes by using the **Target Navigator**. On the command line, you can use the `kill` command. The default mapping of signals to **librcheck** commands is:

Number	Symbol	Command	Description	Equivalent *mallopt()* command
41	SIGRT#0	`control`	Execute a command stored in the file defined by the *MALLOC_CTRL_FILE* variable. This file can contain only one command and it must be in the format given in the `MALLOC_CTRL_CMD` description. The `control` command is a convenient way to execute supported **librcheck** commands, even non-API commands that don't have corresponding signals.	`MALLOC_CTRL_CMD`
42	SIGRT#1	`leaks`	Detect the memory leaks. This is different than enabling the dumping of memory leaks when the program exits. The `leaks` command makes **librcheck** immediately write the data describing all unreferenced blocks at the present time. If leak dumping at exit time is disabled, this command does *not* enable it.	None
43	SIGRT#2	`stop`	Turn off memory tracing. You will stop seeing new allocation and deallocation events in the analysis results.	`MALLOC_TRACING 0`
44	SIGRT#3	`start`	Turn on memory tracing. You will start seeing new allocation and deallocation events in the analysis results.	`MALLOC_TRACING 1`

You can override this mapping by setting *MALLOC_CTRL_SIG*, which **librcheck** checks at startup to set up signal handling.

Page updated: August 11, 2025

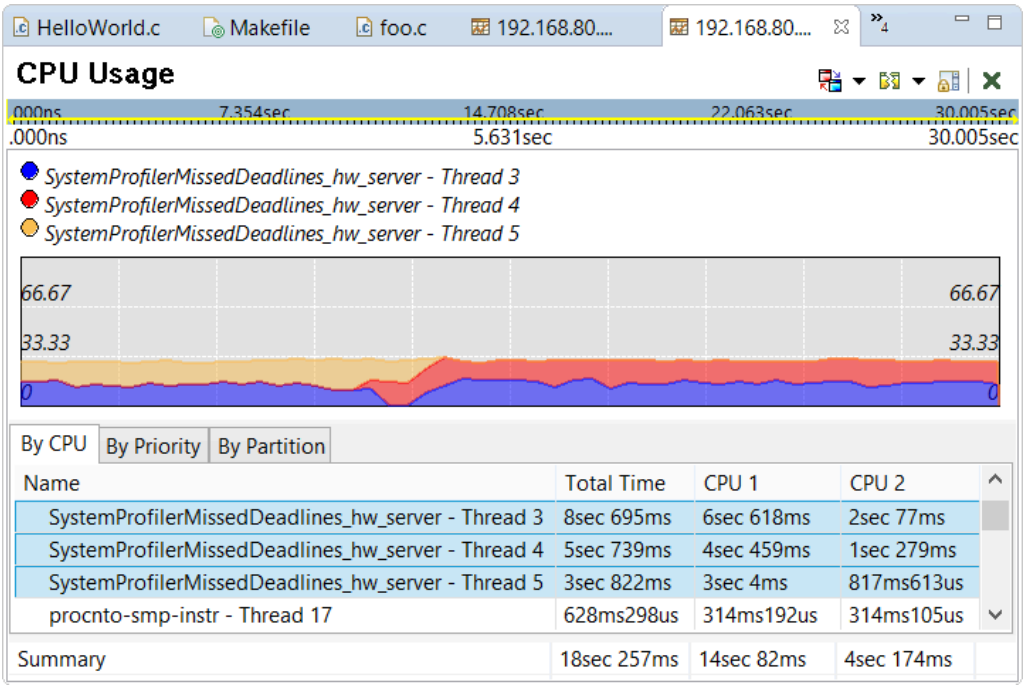
Locating sources of high CPU usage

A key step in optimizing embedded systems is locating sources of high CPU usage. The System Profiler results show you per-process and per-thread CPU consumption over some or all of the trace period so you can spot peaks in the processor loads.

In the Summary pane, the statistics for CPU time breakdown give you an idea of how efficiently your QNX target is performing, by showing you the amount of idle time and the ratio of user time to system time. They also let you spot *interrupt flooding*, which occurs when interrupt time isn't as low as you expect based on your system design. A higher interrupt time could mean faulty hardware, a bad driver, or too many applications being profiled at the same time.

To find periods of high CPU usage, you can start by looking for peaks in the Process & Thread Activity bar graph. These peaks officially indicate high numbers of events at particular times, which often happen when one or more applications are using the CPU heavily. If you run the kernel event trace during an important operational phase, such as system startup or just after a certain application is launched, this graph can tell you when the embedded system is heavily loaded.

Next, to see the CPU resources consumed by individual threads, click the Switch Pane dropdown in the editor controls () and select **CPU Usage**. This pane contains a line graph that illustrates the CPU usage history of any thread selected in the table underneath. The lines and areas representing distinct threads are stacked on each other, so the topmost line indicates the aggregate load of the selected threads. This functionality gives you lots of flexibility, allowing you to see the CPU consumption of any combination of threads that ran during the kernel event trace.



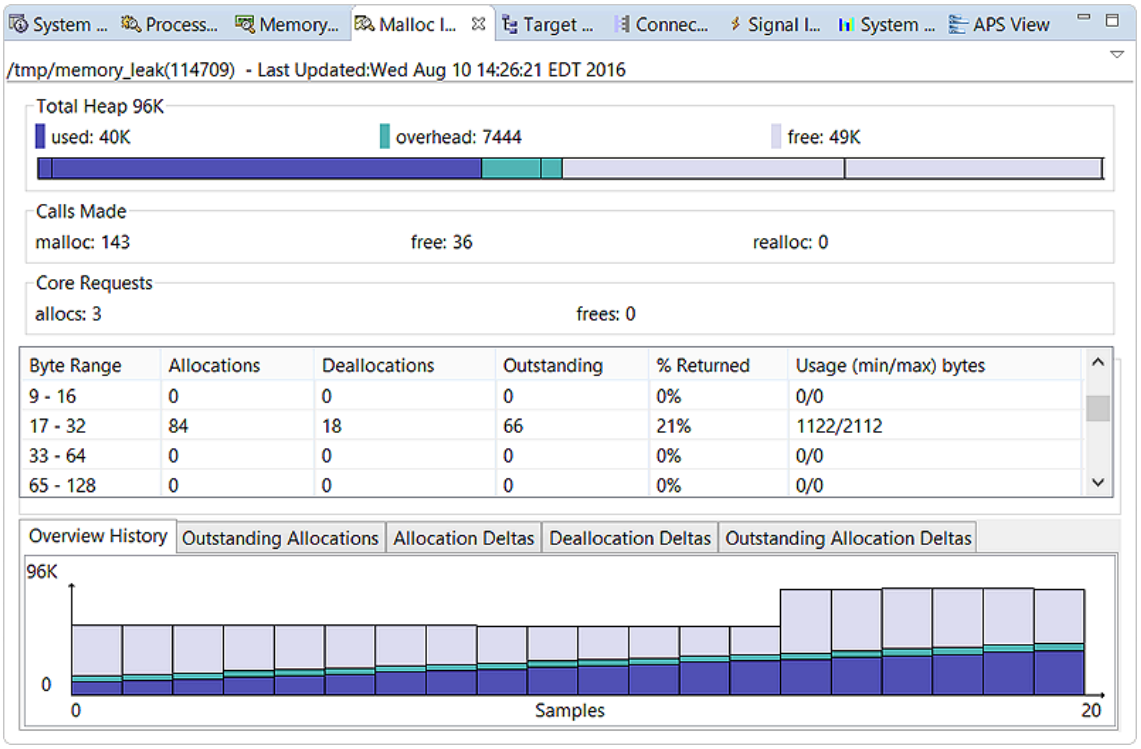
After learning which threads are heavy CPU consumers, you can profile the corresponding applications to see which functions are called most often and have the longest runtimes.

Page updated: August 11, 2025

Malloc Information

You can examine a process's heap through the **Malloc Information** view. This view illustrates heap usage and provides statistics such as allocation call counts and block counts for specific byte ranges.

Leaving this view open lets you monitor a process's heap activity and memory usage. The data come from the general-purpose process-level allocator, so there's no application overhead when the IDE retrieves them. To display the data, the view uses multiple panes, which are described below.



NOTE:

- The screenshot above represents a system targeting SDP 7.1. In an SDP 8.0 target, there's less data available. Continue reading below to understand what's changed.
- All three panes—Overview, Distribution, and Charts—are shown by default but you can hide any of them by deselecting the corresponding options in the dropdown in the upper right corner.

Overview

This top pane provides statistics about activity for the entire heap, in multiple panels.

Total Allocator Managed Memory

This panel contains a bar that indicates the used and free heap space, in color-coded format:

- Used (purple)
- Free (lavender)

This bar graph quickly tells you whether the process is running low on heap space. You can hover the pointer over a section to see a tooltip stating the purpose and size of the corresponding memory area (e.g., used space for small blocks, overhead for metadata of large blocks).

The total heap size is displayed next to the title. Here, the numbers shown are exact measurements that come from the allocation library, unlike with the **Memory Information** and **System Resources** views, which display estimates.

Calls Made

The numbers of *malloc()* and *free()* calls made by the process are given in this panel. The number of *realloc()* calls isn't available in an SDP 8.0 target.

Core Requests

This panel doesn't show statistics in an SDP 8.0 target.

Distribution

This pane doesn't show statistics in an SDP 8.0 target.

Charts

This bottom pane only displays data for the **Overview History**, which represents a timeline of the heap usage snapshots shown in the **Total Allocator Managed Memory** panel. This bar graph automatically rescales as the process increases its total heap size.

Page updated: August 11, 2025

Managing launch configurations

QNX Tool Suite | Integrated Development Environment User's Guide | Developer | Setup

Launch configurations store settings related to program execution, setup, or analysis. When you create a project, the IDE generates a default launch configuration for it. But you can create your own configurations when you need to customize how a project runs and reuse those custom settings.

The IDE provides a wizard to help you create launch configurations. By default, the wizard shows only the QNX launch configuration types, such as **C/C++ QNX Application**. The CDT in the Eclipse platform includes many other launch configuration types, which are explained in **Reference** > (and then)**Run and Debug** in the *C/C++ Development User Guide*. Generally, these aren't meant for use in the QNX development environment, but you can change an IDE preference setting to see and use these other configuration types, as explained in "Creating a non-QNX project". (The same setting affects the configuration types shown in the dialogs accessible through the **Project Explorer** context menu options of **Run As**, **Debug As**, or **Profile As**.)

Here, we explain the QNX launch configuration types, which support launching applications on QNX targets. For convenience, you can access filtered lists to view the launch configurations that support particular launch modes. You can also edit the settings in a configuration, and import configurations to reproduce build and execution conditions for existing applications.

Page updated: August 11, 2025

Analyzing heap memory usage with Valgrind Massif

Valgrind Massif analyzes a program's heap usage by taking heap snapshots as the program runs, then outputs the snapshot data when the program terminates. The IDE visually presents the data by graphing the heap usage over time.

NOTE:
All Valgrind tools can be loaded and run from the command line. However, using the IDE is more convenient because it automates much of the setup by setting Valgrind command options based on UI fields and by copying the analysis results into the host workspace.

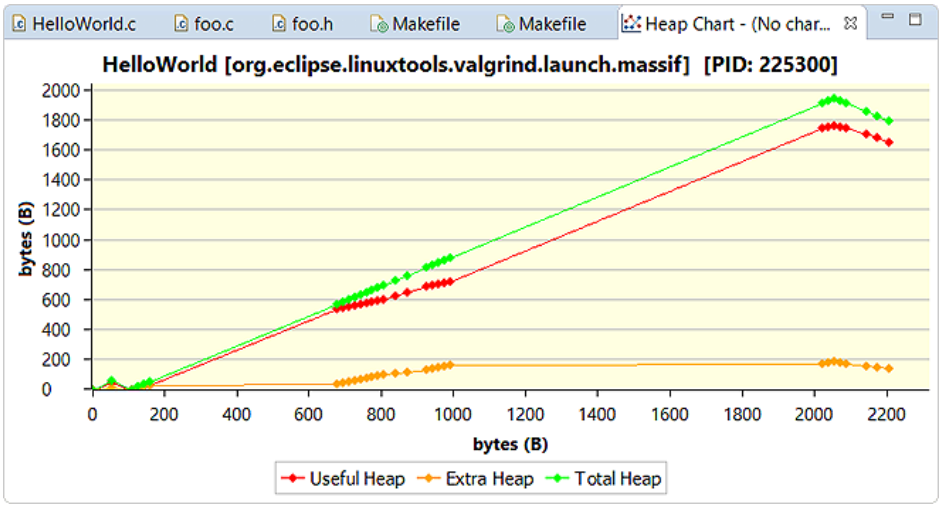
If your target image isn't configured to include the `valgrind` binaries and libraries, the IDE attempts to upload these components when you launch an application with a Valgrind tool enabled. For this to work, the target must have a writable filesystem.

To analyze heap usage with Valgrind Massif:

1. In the launch bar, expand the Launch Configuration dropdown (which is in the middle) and select the project that you want to analyze.
2. In the Launch Target dropdown (on the right), select the target for running your application.
3. In the Launch Mode dropdown (on the left), select Memory.
4. Click the Edit button (⚙) on the right of the Launch Configuration dropdown.
5. In the configuration editor window, access the Valgrind controls by clicking the **Memory** tab on the right and then the Valgrind radio button near the top of this tab.
6. Select **Massif** from the **Tool to run** dropdown.
7. **Optional:** You can change any settings to customize what gets reported in the Valgrind results.
   The **Massif** tab lets you specify how many heap snapshots are taken and how the captured data are presented.
8. Click **OK** to save the configuration changes and close the window.
9. In the launch bar, click the Memory button ( ).

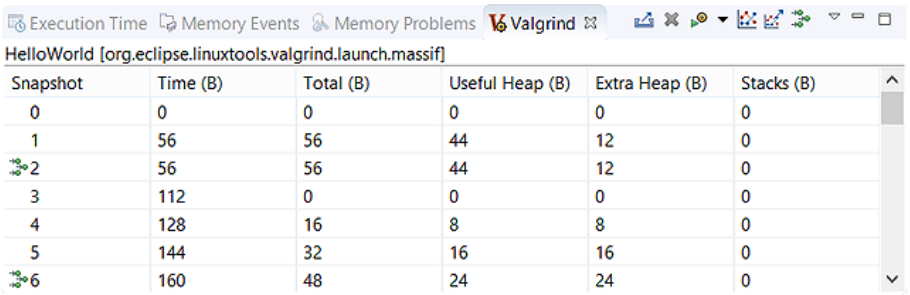
The IDE switches to the QNX Analysis perspective. If necessary, the IDE first builds the binary before uploading it to the target. To analyze the application, the IDE instructs Valgrind to execute the uploaded binary with Massif instrumentation. Then, it creates a session for storing the Valgrind results; this new session is displayed in the **Analysis Sessions** view. When the program terminates, Valgrind writes the results to a log file, which the IDE copies into the directory for the new session.

The memory measurements taken at each heap snapshot are graphed and displayed in a new editor window called **Heap Chart**. The chart drawn is a line graph that illustrates how the application's heap usage changed over time, with diamond-shape points indicating individual snapshots:



The chart plots the heap size along the vertical axis and the execution progress along the horizontal axis. For the execution progress, the metric used is determined by the **Time unit** field.

If you click a point in the chart, the corresponding snapshot is highlighted in the **Valgrind** view. This view displays the heap snapshot data in a table. Each row describes a single snapshot, listing the time measurement and the program's total, useful, and extra heap space at that moment:



Snapshot	Time (B)	Total (B)	Useful Heap (B)	Extra Heap (B)	Stacks (B)
0	0	0	0	0	0
1	56	56	44	12	0
2	56	56	44	12	0
3	112	0	0	0	0
4	128	16	8	8	0
5	144	32	16	16	0
6	160	48	24	24	0

Some heap snapshots are *detailed*, meaning they contain information about where the current blocks were allocated. The rows for these snapshots are indicated with the heap tree icon ( ). Double-clicking in one of these rows displays a tree-like listing of allocation sources, as illustrated and explained in "Finding unused memory with Valgrind Massif".

NOTE:
You can run multiple Valgrind sessions concurrently, using the same tool or different tools, on the same application or different applications. Valgrind log files always contain the PIDs of the Valgrind processes, so their names are always distinct.

Page updated: August 11, 2025

Finding unused memory with Valgrind Massif

Valgrind Massif takes *detailed heap snapshots* to record dynamic memory allocation by program region. The IDE displays this memory breakdown so you can easily find *space leaks*, which occur when the program doesn't free unneeded memory.

NOTE:
All Valgrind tools can be loaded and run from the command line. However, using the IDE is more convenient because it automates much of the setup by setting Valgrind command options based on UI fields and by copying the analysis results into the host workspace.

To find space leaks with Valgrind Massif:

1. In the launch bar, expand the Launch Configuration dropdown (which is in the middle) and select the project in which you want to check for leaks.
2. In the Launch Target dropdown (on the right), select the target for running your application.
3. In the Launch Mode dropdown (on the left), select Check.
4. Click the Edit button ( ) on the right of the Launch Configuration dropdown.
5. In the configuration editor window, access the Valgrind controls by clicking the **Check** tab on the right and then the Valgrind radio button near the top of this tab.
6. Select **Massif** from the **Tool to run** dropdown.
7. **Optional:** You can change any settings to customize what gets reported in the Valgrind results.
   The **Massif** tab lets you specify how often to take detailed heap snapshots and the maximum depth of the allocation trees to report in the results.
8. Click **OK** to save the configuration changes and close the window.
9. In the launch bar, click the Check button ( ).

The IDE switches to the QNX Analysis perspective. If necessary, the IDE first builds the binary before uploading it to the target. To analyze the application, the IDE instructs Valgrind to execute the uploaded binary with Massif instrumentation. Then, it creates a session for storing the Valgrind results; this new session is displayed in the **Analysis Sessions** view. When the program terminates, Valgrind writes the results to a log file, which the IDE copies into the directory for the new session.

The memory measurements taken at each snapshot are graphed in the **Heap Chart** window (for more information, see "Analyzing heap memory usage with Valgrind Massif"). The **Valgrind** view, which is displayed at the bottom, presents the heap snapshot data in a table. The rows for detailed snapshots are indicated with the heap tree icon ( ). Double-clicking in one of these rows displays the heap trees for all detailed snapshots, with the entries for the selected snapshot expanded.

Sometimes, a program might repeatedly allocate blocks and maintain their pointers so they remain accessible but never free these blocks even though their contents aren't needed anymore. In the Valgrind User Manual, this situation is referred to as a "space leak" because it's not a conventional memory leak but the program does waste space. In this manual, we also refer to it as an "implicit leak". To detect this situation, you can look for increasing memory consumption in a specific place indicated in the heap trees, as seen here:

The heap trees should show location information for functions in shared libraries. If you don't see this information, you must manually configure the loading of debug symbols.

Here, the program is accumulating a lot of heap memory at one particular allocation point in *main()*. You would want to review the surrounding code to see if any blocks don't need to be kept in memory for as long. Whether a block is needed depends on your program logic; Massif shows you the evolution of memory usage but you must examine your code closely to learn which blocks can be deallocated.

NOTE:

You can run multiple Valgrind sessions concurrently, using the same tool or different tools, on the same application or different applications. Valgrind log files always contain the PIDs of the Valgrind processes, so their names are always distinct.

Page updated: August 11, 2025

Measuring code coverage

QNX Tool Suite | Integrated Development Environment User's Guide | Developer | Setup

You can build and configure projects in the IDE to collect and display code coverage measurements as the program runs.

You can also run a binary built with code coverage instrumentation from the command line.

Code coverage can be measured for any executable binary, whether it's a test program or an application. For applications, you can monitor which areas of code are frequently executed and hence, are the best ones to optimize. However, the main use of the Code Coverage tool is to measure the quality of test programs—if any code isn't exercised during testing, it could contain hidden bugs. You can use the results to determine which areas aren't being covered and then write new test cases to cover them.

CAUTION:
The Code Coverage tool can make a program behave unpredictably and incorrectly. When the IDE requests coverage data by sending signals to `qconn`, signals are also delivered to blocking function calls, which might change program behavior. For instance, a *sleep()* call might return too soon if the IDE requests coverage data during the timeout period.

To measure code coverage from the IDE:

1. Define the necessary build settings.
   You must add certain flags to the compiling and linking commands. The way to do this depends on your makefile type, as explained in "Enabling Code Coverage instrumentation".
2. In the launch bar, expand the Launch Configuration dropdown and select the project for which you want to measure code coverage.
3. In the Launch Mode dropdown, select Coverage.
4. In the Launch Target dropdown, select the target for running the program.
5. **Optional:** To customize how coverage data are collected, click the Edit button (⚙) on the right of the Launch Configuration dropdown, then click the Coverage tab (which is the rightmost tab).

   You can adjust how often the IDE signals `qconn` to request coverage data or set a nondefault target location for writing the **.gcda** data files. Details about all tool settings are given in "How to configure Code Coverage".

   When you're finished configuring the tool, click **OK** to save the changes and close the window.
6. In the launch bar, click the Coverage button ( ).

The IDE switches to the QNX Analysis perspective. If necessary, it first builds the binary (with code coverage instrumentation). Then, the IDE uploads the binary and starts running it on the target. When this happens, a new Code Coverage session is created and displayed in the **Analysis Sessions** view. For details on interpreting the results, see "How Code Coverage results are presented".

NOTE:
You can run multiple sessions concurrently—the **Analysis Sessions** view lets you interact with any number of active sessions.

Generating code coverage data from the command line

QNX Tool Suite | Integrated Development Environment User's Guide | Developer | Setup

After building an application with code coverage instrumentation, you can copy its binary to a QNX target and run it from the command line. The generated coverage data contain the same measurements reported through the IDE.

To measure code coverage from the command line:

1. Define the necessary build settings.
   You must add certain flags to the compiling and linking commands. The way to do this depends on your makefile type, as explained in "Enabling Code Coverage instrumentation".
2. Build the application and verify that the correct binary files are generated.

In addition to the coverage variant of the executable binary (which is noticeably larger than the debug variant), you should find notes (**.gcno**) files for all object files. Notes files are needed for later viewing the coverage results in the IDE.

3. On the target, set the *GCOV_PREFIX* environment variable based on where you want to store the coverage data (**.gcda**) files.

   The files are written to a subdirectory that matches the host location where the binary was built. This subdirectory is found in the path specified by *GCOV_PREFIX*. For instance, if the binary was built in **/home/*workspace_dir*/*project_name*/x86_64/o-g** on the host and *GCOV_PREFIX* is set to **/usr/coverage**, then the data files are written to **/usr/coverage/home/*workspace_dir*/*project_name*/x86_64/o-g/** on the target.

   If the variable isn't defined, the program tries to write the files to a similar but absolute path (e.g., **/home/*workspace_dir*/*project_name*/x86_64/o-g/**). But if the root path is non-writable (as by default), then the directory can't be created and hence, a profiling error message is displayed:

   ```
   profiling:/home:Cannot create directory,
   profiling:/home/workspace_dir/project_name/x86_64/o-
   g/some_module.gcda
   ```

4. Run the application.
   No specific command-line options are needed to enable code coverage because the instrumentation code outputs coverage results as the program runs.

When the application exits normally, you should see a data file for each source file, at the location determined by *GCOV_PREFIX*. Note that if a serious error occurs and the application can't exit cleanly, no data files are generated.

Page updated: August 11, 2025

Attaching Memory Analysis to a running process

If you want to start analyzing heap activity or finding memory errors for a running process, you can attach the Memory Analysis tool. The IDE then displays the **librcheck** data received from the target, to illustrate the process's heap allocations and list any memory problems.

NOTE:
The application that you want to analyze must have been run with **librcheck**. Also, your host machine must have an IP connection to your target machine.

To attach Memory Analysis to a running process:

1. In the launch bar, expand the Launch Configuration dropdown (which is in the middle) and select the application for which you want to see memory data.
2. In the Launch Target dropdown (on the right), select the target on which the application is running.
3. In the Launch Mode dropdown (on the left), select Attach.
4. Click the Edit button (⚙) on the right of the Launch Configuration dropdown.
5. Enable Memory Analysis in the launch configuration:
    a. Click the **Tools** tab on the right.
    b. Click the Memory Analysis radio button.
    c. **Optional:** If needed, configure any tool settings to customize what gets reported.
    d. **Optional:** Enable or disable other analysis tools.
    You can disable the debugger if need be, by unchecking the box at the top of the **Debug** tab.

    CAUTION:
    We recommend that you don't run Memory Analysis at the same time as the GDB debugger because the setup needed to prevent the program from crashing is more complicated. If you have to run both tools concurrently, you must follow the steps given in "Running Memory Analysis and the GDB Debugger concurrently" instead of the remaining steps given here.

    You can also run the System Profiler (by checking this tool's box at the bottom of the **Tools** tab), to perform a kernel event trace while gathering memory data.
6. When you've finished configuring the tools, click **OK** to save the changes and exit the window.
7. Click the Attach button ( ).
The IDE switches to the Debug perspective if the debugger is enabled, or the QNX Analysis perspective if it's not, then opens the **Select Process** popup window. This window lists the processes with the same name as the binary specified in the **Main** tab.
8. Click the process that you want to attach to, then click **OK**.

The IDE attaches Memory Analysis and any other enabled tools to the selected process. In the current perspective, you'll see a new session for storing the debugging or analysis results. Depending on the active tools and their configuration, these results might get updated as the program runs.

In the QNX Analysis perspective, you can see the **librcheck** statistics displayed by the Memory Analysis tool. The Memory Analysis reference explains how to interpret the results.

Page updated: August 11, 2025

Finding memory corruption with Memory Analysis

You can enable error checking for Memory Analysis and launch an application to find cases of memory corruption, such as bad pointer values given to C library functions as well as out-of-bounds data writes. The Memory Analysis tool detects this from the data sent by the debug allocation library (**librcheck**).

> **NOTE:**
> You can run an application that uses **librcheck** from the command line. However, using the IDE is more convenient because the Memory Analysis tool automates the requesting of data from this memory-tracing library and presents the results in a convenient format.

To find memory corruption with Memory Analysis:

1. In the launch bar, expand the Launch Configuration dropdown (which is in the middle) and select the project for which you want to check memory.
2. In the Launch Target dropdown (on the right), select the target for running your application.
3. In the Launch Mode dropdown (on the left), select Check.

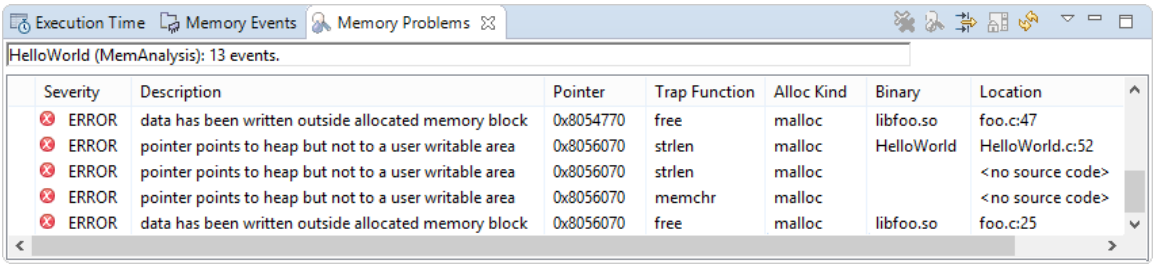
   > **NOTE:**
   > Although you can select Debug mode and manually enable Memory Analysis in the launch configuration, we recommend that you don't run this tool at the same time as the GDB debugger because the setup is more complicated. If you absolutely have to run both tools concurrently, you must follow the steps given in "Running Memory Analysis and the GDB Debugger concurrently" instead of the remaining steps given here.

4. To confirm that Memory Analysis is the active tool or to adjust how it checks for memory errors, you must examine the launch configuration. To do this:
   a. Click the Edit button (⚙) on the right of the Launch Configuration dropdown.
   b. In the configuration editor window, click the **Check** tab on the right to access the tool controls.
   c. Ensure that the Memory Analysis radio button is selected, then uncheck the **Record memory allocation/deallocation events** box in the **Memory Tracing** panel.
   d. In the **Memory Errors** panel, you can enable or disable specific memory checks based on what types of problems you want to look for.
   e. Click **OK** to save the configuration changes and close the window.
5. In the launch bar, click the Check button ( ).

The IDE switches to the QNX Analysis perspective. If necessary, it first builds the binary. Then, the IDE uploads the binary and starts running it on the target, with **librcheck** loaded. When this happens, a new session is created for storing the data produced by the **librcheck** library; this session is displayed in the **Analysis Sessions** view.

The Memory Analysis tool lists corruption and other memory errors in the **Memory Problems** view, which is displayed at the bottom. By default, errors for the entire program are listed but you can expand the **Analysis Sessions** entry and click a program component (e.g., process, thread) to show only its errors.

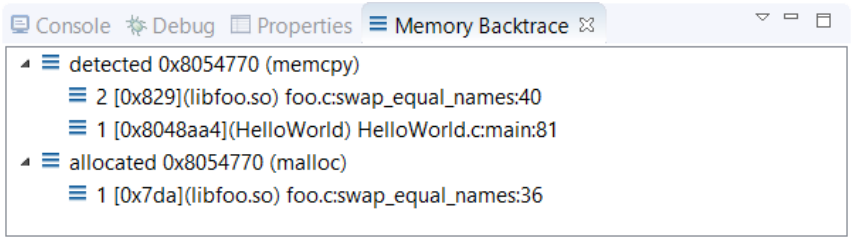
In **Memory Problems**, all corruption errors contain a red X next to the ERROR keyword and also provide an error description and the name of the function that trapped the error. When Memory Analysis knows the exact location where memory was corrupted, the binary object, source file, and line of code are also given. If the tool can't determine the corruption location (because the error was caught in another part of the program), it can't provide those details.



For shared libraries, you should see location information for errors that occurred in their functions. If you don't see this information, you must manually define the library paths. The Memory Analysis reference explains all error fields and messages.

You can click an error to see stack traces of the code that allocated the memory and the code that detected its corruption. These stack traces are shown in the **Memory Backtrace** view displayed in the lower left area. The depth of the "detected" call chain is restricted by the **Limit back-trace depth to** field found in the Memory Errors panel. The depth of the "allocated" call chain is based on the field with the same name but found in the Memory Tracing panel.

In the following screenshot, the depth is set to 2 for detection traces and to 1 for allocation traces:



If you double-click an error in **Memory Problems** or a stack frame in **Memory Backtrace**, the IDE opens the source file at the indicated line (when source file information is available). This feature lets you quickly find where a memory block was allocated or corrupted.

NOTE:
You can run multiple Memory Analysis sessions concurrently, on the same application or different applications, but you must specify distinct file or device output paths for these sessions in the Advanced settings.

Page updated: August 11, 2025

Finding memory leaks with Memory Analysis

The **librcheck** library tracks all memory operations and references, which lets it detect leaked blocks. Memory Analysis can request leak checks from **librcheck** at regular intervals and at program exit time.

> **NOTE:**
> You can run an application that uses **librcheck** from the command line. However, using the IDE is more convenient because the Memory Analysis tool automates the requesting of data from this memory-tracing library and presents the results in a convenient format.

To find memory leaks with Memory Analysis:

1. In the launch bar, expand the Launch Configuration dropdown (which is in the middle) and select the project in which you want to check for leaks.
2. In the Launch Target dropdown (on the right), select the target for running your application.
3. In the Launch Mode dropdown (on the left), select Check.

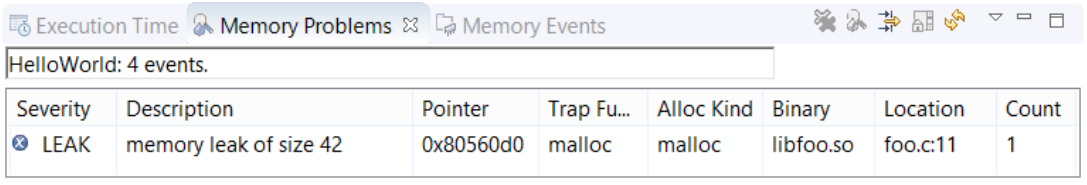
   > **NOTE:**
   > Although you can select Debug mode and manually enable Memory Analysis in the launch configuration, we recommend that you don't run this tool at the same time as the GDB debugger because the setup is more complicated. If you absolutely have to run both tools concurrently, you must follow the steps given in "Running Memory Analysis and the GDB Debugger concurrently" instead of the remaining steps given here.

4. **Optional:** To confirm that Memory Analysis is the active tool or to adjust how it checks for memory leaks, you must examine the launch configuration. To do this:
   a. Click the Edit button (⚙) on the right of the Launch Configuration dropdown.
   b. In the configuration editor window, click the **Check** tab on the right to access the tool controls.
   c. Ensure that the Memory Analysis radio button is selected. In the fields underneath, you can enable memory leak checks and set their frequency.
   d. Click **OK** to save the configuration changes and close the window.
5. In the launch bar, click the Check button ( ).

The IDE switches to the QNX Analysis perspective. If necessary, it first builds the binary. Then, the IDE uploads the binary and starts running it on the target, with **librcheck** loaded. When this happens, a new session is created for storing the data produced by the **librcheck** library; this session is displayed in the **Analysis Sessions** view.

The Memory Analysis tool lists leaks and other memory errors in the **Memory Problems** view and memory management operations in the **Memory Events** view, which are displayed at the bottom. By default, heap activity for the entire program is listed but you can expand the **Analysis Sessions** entry and click a program component (e.g., process, thread) to show only its errors and operations.

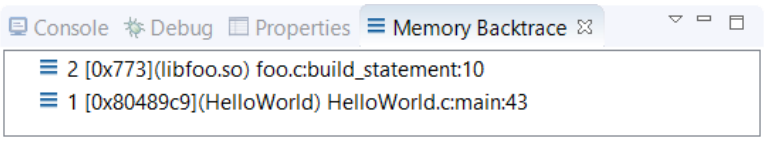
In **Memory Problems**, all leaks have a blue X next to the `LEAK` keyword and a summary with the leak size (in bytes). This is followed by the address of the lost block, the trap function (which is always *malloc()*), and the location where the block was allocated:



For shared libraries, you should see location information for lost blocks allocated in their functions. If you don't see this information, you must manually define the library paths.

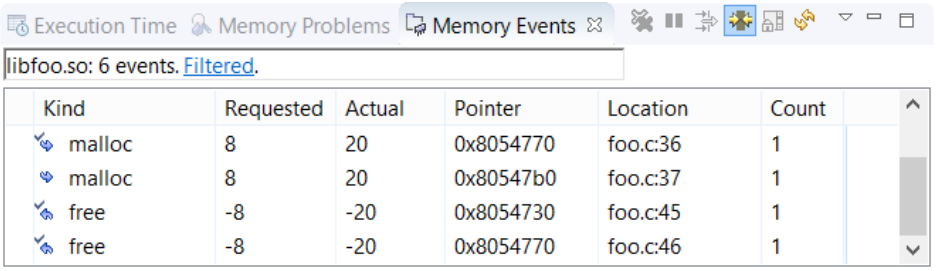
You can click a leak to see the stack trace of the memory allocation point. This stack trace is shown in the **Memory Backtrace** view displayed in the lower left area. The depth of the "allocated" call chain is restricted by the **Limit back-trace depth to** field in the Memory Tracing controls.

In the following screenshot, the depth is set to 2 for allocation traces:



If you double-click a leak in **Memory Problems** or a stack frame in **Memory Backtrace**, the IDE opens the source file at the indicated line (assuming the backtrace depth is greater than 0). This feature lets you quickly find where a particular memory block was allocated.

In **Memory Events**, you can review the list of allocations and deallocations (frees) and see when leaks may have occurred within the sequence of memory operations:



In the `Kind` column, the arrow icon next to the function name points either right (for allocations) or left (for deallocations). When the icon has a checkmark, there's a matching request. Thus, *malloc* events without a checkmark *may* indicate leaks. Depending on your program's design or behavior, many of these items might be not explicit but rather implicit leaks. The Memory Analysis reference explains all event fields.

Above the events list is a summary field that lists the name of the selected component, the number of events, and a link with the text `Filtered`. Clicking this link opens the **Memory Events Filter**, in which you can choose to hide all events related to blocks for which there are matching allocations and deallocations. This feature lets you filter out all non-leak events.

NOTE:
You can run multiple Memory Analysis sessions concurrently, on the same application or different applications, but you must specify distinct file or device output paths for these sessions in the Advanced settings.

Page updated: August 11, 2025

Analyzing heap memory usage with Memory Analysis

You can launch an application with Memory Analysis enabled to observe the heap activity over the application's lifetime. The Memory Analysis tool reads the data sent by the debug allocation library (**librcheck**) and displays graphs and event information describing heap allocations and deallocations.

> **NOTE:**
> You can run an application that uses **librcheck** from the command line. However, using the IDE is more convenient because the Memory Analysis tool automates the requesting of data from this memory-tracing library and presents the results in a convenient format.

To analyze heap usage with Memory Analysis:

1. In the launch bar, expand the Launch Configuration dropdown (which is in the middle) and select the project that you want to analyze.
2. In the Launch Target dropdown (on the right), select the target for running your application.
3. In the Launch Mode dropdown (on the left), select Memory.

   > **NOTE:**
   > Although you can select Debug mode and manually enable Memory Analysis in the launch configuration, we recommend that you don't run this tool at the same time as the GDB debugger because the setup is more complicated. If you absolutely have to run both tools concurrently, you must follow the steps given in "Running Memory Analysis and the GDB Debugger concurrently" instead of the remaining steps given here.

4. **Optional:** To confirm that Memory Analysis is the active tool or to adjust how it measures and reports heap usage, you must examine the launch configuration. To do this:
   a. Click the Edit button (⚙) on the right of the Launch Configuration dropdown.
   b. In the configuration editor window, click the **Memory** tab on the right to access the tool controls.
   c. Ensure that the Memory Analysis radio button is selected. In the fields underneath, you can set the stack trace and heap snapshot settings based on what you want to see in the results.
   d. Click **OK** to save the configuration changes and close the window.
5. In the launch bar, click the Memory button ( ).

The IDE switches to the QNX Analysis perspective. If necessary, it first builds the binary. Then, the IDE uploads the binary and starts running it on the target, with **librcheck** loaded. When this happens, a new session is created for storing the data produced by the **librcheck** library; this session is displayed in the **Analysis Sessions** view.

The IDE also opens an editor window that shows the program's heap memory activity. The **Allocations** tab is selected by default and it displays two charts that illustrate the allocation and deallocation sizes for all events and for a selected subset of events.

The editor tabs are explained in full in "How to use the Memory Analysis editor".

> **NOTE:**
> You can run multiple Memory Analysis sessions concurrently, on the same application or different applications, but you must specify distinct file or device output paths for these sessions in the Advanced settings.

Page updated: August 11, 2025

Memory Analysis

Memory Analysis is a QNX tool that uses the debug version of the memory allocation library (**librcheck**) to track heap memory, validate pointer arguments to C library functions, and detect memory corruption. This tool displays data received from the **librcheck** library about memory events and problems that occur in an active application.

How to configure Memory Analysis

You can enable the Memory Analysis tool by selecting Memory as the launch mode. Although this mode supports other integrated tools, Memory Analysis is selected by default. Aside from Memory mode, you can select any other mode except Coverage and enable this tool by opening the launch configuration and clicking the Memory Analysis radio button in the rightmost tab.

The tool settings are made visible in four dropdown panels:

- **Memory Errors**
- **Memory Tracing**
- **Memory Snapshots**
- **Advanced Settings**

Memory Errors

Enable error detection
: Enables or disables memory error-checking; you can disable this feature but retain its settings

Verify parameters in string and memory functions
: When enabled (as by default), makes Memory Analysis verify pointers passed to functions such as *memcmp()* and *strcpy()* and report bad values (including NULL) as errors

Enable bounds checking (where possible)
: Whether to check for data being written before the beginning or after the end of allocated blocks (default: enabled). This checking is done only for dynamically allocated blocks.

Enable check on *realloc()*/*free()* argument
: Whether to check the pointer argument passed to *realloc()* or *free()* (default: enabled)

When an error is detected
: The action to take when a memory error is detected. The default setting is `report the error and continue` but you can also choose `launch the debugger` or `terminate the process`.

Limit back-trace depth to
: Specifies how many frames up the stack the tool should look when reporting memory corruption errors. If you set this field to 1, Memory Analysis lists only the line where the error occured. A value of 2 makes the tool list the error line and the line in the calling function. A greater depth creates greater overhead, so there's a trade-off between performance and the level of stack data reported. A value of 0 disables memory error stack tracing.

NOTE:
For **librcheck** to produce stack trace data, your target system must include **libunwind.so.8** in a location accessible to *dlopen()*. For more information, see the *dlopen()* entry in the *C Library Reference*. The **librcheck** library still functions without **libunwind**, but prints a warning that it can't produce backtraces when tracing function calls.

Perform leak check every
: The time interval, in milliseconds, to check for memory leaks. Leak checks add overhead, so you must pick an interval value based on how quickly you want to detect leaks versus how much overhead you can tolerate. You can set a value of 0 to disable leak checking.

  For this option to work, the control thread must be enabled in the Advanced Settings.

Perform leak check when process exits
: When enabled (as by default), makes Memory Analysis check for memory leaks when the process exits. Some leaks can be caught only if this option is enabled. For this option to work, the application must exit "cleanly" by calling *exit()* or in the *main()* function, *return*.

Memory Tracing

Record memory allocation/deallocation events

Enables or disables memory tracing; you can disable this feature but retain its settings

Limit back-trace depth to

Specifies how many frames up the stack the tool should look when reporting memory allocation traces. If you set this field to 1, only the exact line where memory was allocated is shown. A value of 2 shows the allocation line and the line in the calling function. A greater depth creates greater overhead, so there's a trade-off between performance and the amount of stack data reported. A value of 0 disables memory allocation tracing.

NOTE:
For **librcheck** to produce stack trace data, your target system must include **libunwind.so.8** in a location accessible to *dlopen()*. For more information, see the *dlopen()* entry in the *C Library Reference*. The **librcheck** library still functions without **libunwind**, but prints a warning that it can't produce backtraces when tracing function calls.

Minimum allocation to trace

The minimum size, in bytes, for a block of memory to be included in the allocation stack trace results. A value of 0 (the default) means there's no filtering of small blocks.

Maximum allocation to trace

The maximum size, in bytes, for a block of memory to be included in the allocation stack trace results. A value of 0 (the default) means there's no filtering of large blocks.

Memory Snapshots

Memory Snapshots

Enables or disables memory snapshots; you can disable this feature but retain its settings

Perform snapshot every

The periodic time interval (in milliseconds) for taking heap snapshots. Each snapshot captures details about the full block list of the heap. Performing regular snapshots lets you see how the heap contents change over time. The default value is 50 000 milliseconds.

Bins counters

A comma-separated list specifying the sizes, in bytes, for defining the bins (groups) when reporting the allocation counts in the **Bins** tab. Suppose you fill in this field with `32, 256`. The heap snapshot results will then contain separate counts for memory blocks up to 32 bytes in size, between 32 and 256 bytes, and more than 256 bytes.

If you leave this field blank, the default block size boundaries are powers of 2, from 2 B up to 4 KB, with an additional upper boundary of 2 GB to cover large blocks.

Advanced Settings

Runtime library

The filename of the allocation library to use to collect memory data. You would change this field only when working with an older SDK version and using a nondefault library.

Use regular file / Use streaming device

Whether to output the data to a regular file or a streaming device. By default, **Use regular file** is selected. In this case, the `qconn` service can't become blocked when writing data, but there's a file size limit of 2G. This setting is handy for postmortem analysis.

When **Use streaming device** is selected, the data is streamed directly to the IDE using a device path created by `qconn`. There's no size limit on the data that can be logged, but `qconn` becomes blocked if it sends data faster than the IDE can read it.

Target output file or device

Path of the file or device on the target for outputting the memory data. When **Use regular file** is selected, this field defaults to **/tmp/traces.rmat**.

When **Use streaming device** is selected, this field is set to **/dev/rcheck/traces.rmat**.

Create control thread

Create a thread to handle data requests at runtime. This setting is enabled by default and is required for doing regular leak checks.

Use dladdr to find dll names

This field isn't used when you're working with the SDK from QNX SDP 7.0 or later, or the **librcheck.so** runtime library, so it's unavailable by default. If you switch to an older SDK version and enter **libmalloc_g.so** in the **Runtime library** field, this checkbox becomes available. Then, you must check the **Use dladdr** field if you want to see backtrace information from shared objects (dynamically linked libraries) built with debugging information.

Show debug output on console

Show the **librcheck** output in the **Console** view (default: disabled)

NOTE:

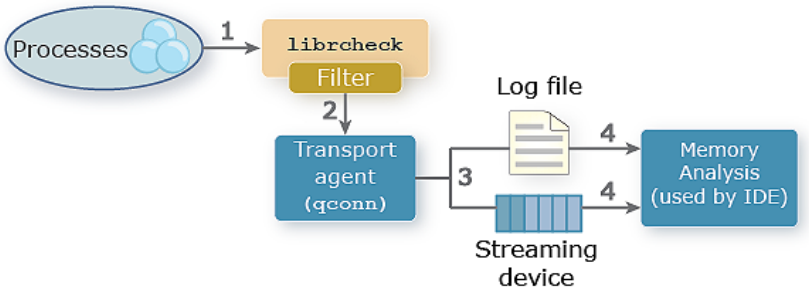
Below the four dropdown panels, the **Configure Shared Libraries Paths** link takes you to the **Libraries** tab. In this tab, you must define the path of any shared library for which you want to see symbol information in the Memory Analysis results.

How the tool works

When Memory Analysis has been enabled in the launch configuration, the application uses the debug version of the allocation library, **librcheck**. This library version tracks the history of all dynamic memory blocks and generates data about where in the program each block was allocated or freed. It also contains its own implementations of memory- and string-related functions, such as *memcmp()* and *strcpy()*, so it can validate pointer arguments passed to them and report any invalid pointers.

The **librcheck** library sends data about the memory activity of application processes to the **qconn** agent, which then writes the data to a log file or a streaming device. Within the IDE, the Memory Analysis tool reads and displays the analysis data:

Figure 1Memory Analysis: data flow



The data flow for Memory Analysis works as follows:

1. **Reacting to process memory activity**

   Whenever an application process allocates or frees memory, or calls any function implemented in **librcheck**, the library generates data describing the memory activity.

2. **Sending data to the transport agent**

   The **librcheck** library doesn't interact directly with the IDE on the host. Like other runtime analysis components, **librcheck** must send its data to a transport agent. In this example, we show the `qconn` service, but you could use any service that talks to the IDE over an IP connection.

3. **Logging data to an output file or device**

   The `qconn` agent outputs the memory data to either a local file or a streaming device, depending on the advanced settings.

4. **Reading and presenting the data**

   Within the IDE, the Memory Analysis tool reads the memory data from the log file or streaming device, then visually presents the data in a new analysis session.

How Memory Analysis results are presented

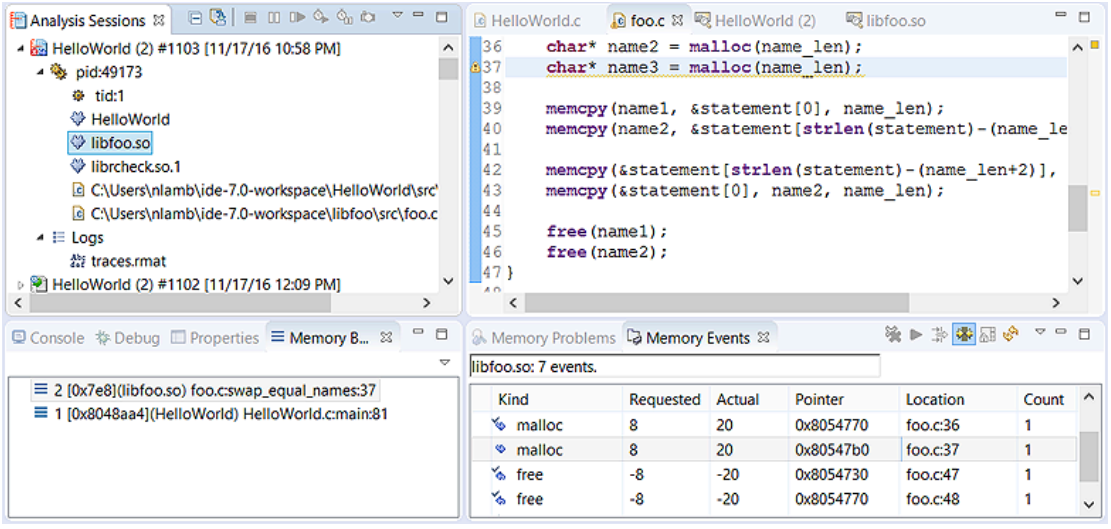
When you launch an application with Memory Analysis enabled, the IDE switches to the QNX Analysis perspective and opens the Memory Analysis editor, which displays charts illustrating the application's heap usage. In the **Analysis Sessions** view, the IDE creates a new session for storing the analysis results. Each open Memory Analysis session has a header containing the tool's open session icon (), the binary name, the session number, and the launch time.

NOTE:

In this IDE version, the latest session is displayed at the top of the view, not at the bottom like in previous versions.

Under the header, all processes in the program being analyzed are listed. You can expand each process entry to see the threads, the executable binary, any shared libraries, and all source files that run within that process. For source files compiled into the executable binary, they're shown only if the binary was built with debug information. For source files compiled into shared libraries, you must have a debug version of these libraries within the shared libraries path on the host to see them listed.

When you click a program component (e.g., process, thread) within a session, the **Memory Events** and **Memory Problems** views display details about the memory management operations and any errors that occurred within that component. You can then click a line in these other two views to see the allocation or deallocation call chain in the **Memory Backtrace** view. Double-clicking a program component opens another editor window that shows the heap usage for that component. Double-clicking a call chain entry opens the source file to the corresponding line of code (when source file information is available):



Each session also has a `Logs` entry that lists the filename (without the path) specified in the **Target output file or device** field. If you double-click this item, the IDE opens the log containing the analysis data gathered so far. This action is handy when you're working with QNX customer support and you need to look up something specific. When the session ends, you'll see the trace (**.rmat**) file stored in ***workspace_dir*/.metadata/.plugins/com.qnx.tools.ide.common.sessions.core/sessions/*session_number*/**. You can then import these analysis results to view them later.

Finally, you can reopen a closed session (which is indicated with a different icon,) by double-clicking its **Analysis Sessions** entry. The IDE then redisplays the program components for this session in that same view and the heap usage charts in the editor pane.

Error details and messages

Each table row in **Memory Problems** describes one error. The default columns, from left to right, are:

- **Severity** – Either `Error`, for memory read and write errors, or `Leak`, for leaks
- **Description** – Informative message summarizing the error
- **Pointer** – Address of the memory block involved
- **Trap Function** – Function that trapped the error
- **Alloc Kind** – Type of allocation used for the associated block
- **Binary** – Executable or library file where the error occurred
- **Location** – Source file and line of code where the error occurred. For this field to be meaningful, debug symbols must be available for the file.

You can change which details get displayed by clicking the dropdown button () in the upper right corner of the view and choosing **Preferences**. This action opens a popup window that lets you choose the columns to display and the order to list them, from left to right.

In the **Description** column, the possible error messages and their meanings are:

Error Message	Meaning
`pointer does not point to heap area`	The program attempted to free non-heap memory.

Error Message	Meaning
`data has been written outside allocated memory block`	The program attempted to write data to a region beyond the allocated memory.
`data area is not in use (can't be freed or realloced)`	A buffer overflow occurred in the heap and it's now corrupted.
`unable to get additional memory from the system`	There is no more heap memory that can be allocated.
`pointer points to the heap but not to a user writable area`	A buffer overflow occurred in the heap and it's now corrupted.
`free'd pointer isn't at start of allocated memory block`	The program attempted to deallocate a pointer that shifted from its original value returned by the allocator.
`memory leak of size n`	A heap block of size *n* has been lost.

Event details

Each table row in **Memory Events** describes one event. The default columns, from left to right, are:

- **Kind** – Type of memory operation (malloc, calloc, new, free, etc.). The arrow icon points right for allocations and left for deallocations. When the icon has a checkmark, there's a matching request. For example, an allocation with a checkmark has a corresponding free.
- **Requested** – Number of bytes requested
- **Actual** – Number of bytes actually allocated or deallocated
- **Pointer** – Address of the memory block involved
- **Location** – Source file and line of code where the event occurred. For this field to be meaningful, debug symbols must be available for the file.
- **Count** – Number of individual allocations (this is more than 1 only when they're grouped)

You can change which details get displayed by clicking the dropdown button (▽) in the upper right corner of the view and choosing **Preferences**. This action opens a popup window that lets you choose the columns to display and the order to list them, from left to right.

How to use the Memory Analysis editor

When you launch a Memory Analysis session, the IDE automatically opens an editor window that shows the analysis results for that session. These results are refreshed regularly as the program runs. If you double-click any component within a session, the IDE opens another window to display only the results for that component. You can open such windows for both active and finished sessions.

The editor window contains two tabs:

- **Allocations**
- **Settings**

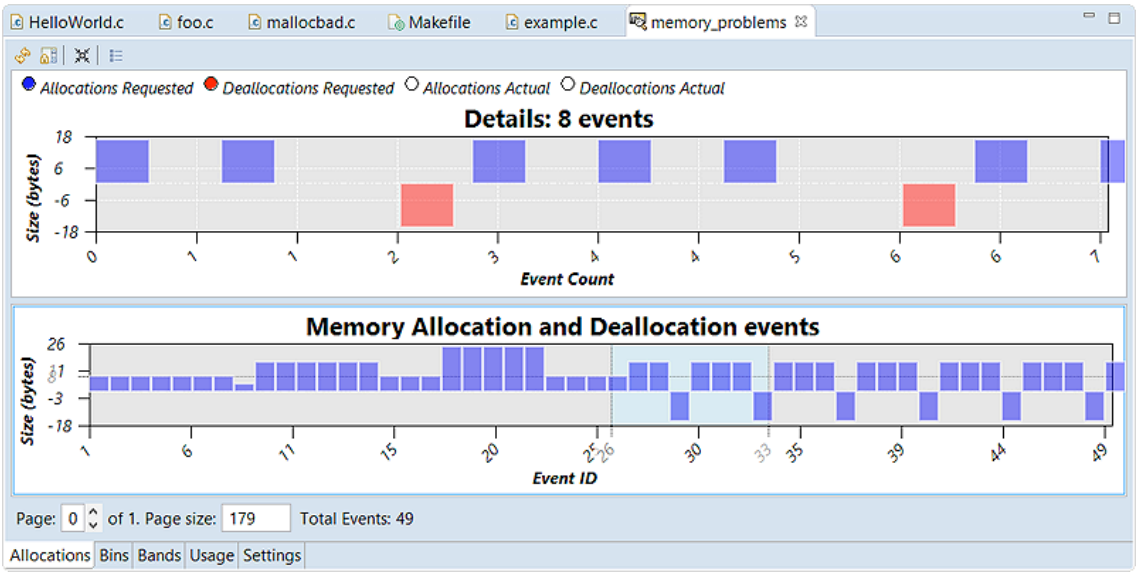
Windows showing program-level data display both tabs. Windows displaying data for a program component display only the **Allocations** tab.

Allocations tab

This tab is shown when the editor is first opened, and it provides an interactive display of events related to dynamic memory and lets you view details about a subset of events. Two charts are displayed:

- **Details** – In the top part of the editor, this chart illustrates the byte sizes of the memory blocks related to the events selected in the bottom part.
- **Memory Allocation and Deallocation Events** – In the bottom part of the editor, this chart provides an overview of memory events.

To select a subset of events, left-click and drag the mouse within the bottom chart. When you release the mouse button, the top chart is redrawn to graph the details for the newly selected events. The byte size range is indicated along the vertical axis and the event counts (i.e., their ordering within the selected event subset) is indicated along the horizontal axis. In the bottom chart, the startpoint and endpoint of the selection are indicated:



> **NOTE:**
> The screenshot above represents a system targeting SDP 7.1. In an SDP 8.0 target, you won't see data for the **Bins**, **Bands**, or **Usage** tabs.

The Details chart uses a 2D bar chart format by default. You can change the format by right-clicking anywhere in the top part of the editor, choosing **Chart types**, then choosing one of these four options: **BarChart** (the default), **BarChart_3D**, **Differentiator**, and **Differentiator_3D**.

The radio buttons just above the Details chart determine which types of events are graphed. By default, requested allocations and deallocations are displayed, but you can also display actual allocations and actual deallocations to see how much the heap memory footprint grows or shrinks.

The Memory Allocation and Deallocation Events chart illustrates the byte sizes for memory events too, but for a larger section of the session data set. In this bottom chart, the byte size range is shown along the vertical axis and by default, the event ID range is shown along the horizontal axis.

You can right-click anywhere in the bottom part of the editor and select **By Timestamp** to display the time range (in microseconds) instead of the event ID range (and you can return to the event ID labelling by selecting **By Event ID** from this same menu). This context menu also has other options:

Filters
: Open the Memory Events Filter window so you can set fields for filtering the events shown in the two charts

Zoom In
: After you've selected a subset of events, click this option to zoom in on those events

Zoom Out
: After zooming in on a subset of events, click this option to zoom out again

Show Events Table
: Update the **Memory Events** view to display information about the selected subset of events

Initially, the data set from the entire session is graphed. Because the bottom chart can get quite wide if there's a lot of data, you can divide the chart into different “pages” to display only part of the data set at a time. To do this, in the **Page Size** text field just below the chart, enter the number of events that you want to see at a given time. On the left of this text field is a spinner that lets you manually enter a page to display, or navigate between pages with the arrow buttons.

The **Allocations** tab provides a toolbar (in the top left corner) that supports the following actions:

- Reload ( ) – Load the latest analysis results and redraw the charts to show them; this action is meaningful only for active sessions
- Prevent Auto-refresh ( ) – For active sessions, disable the automatic updating of the chart
- Toggle Overview Chart ( ) – Toggle the visibility of the bottom chart
- Show Events Table ( ) – Update the **Memory Events** view to display information about the selected subset of events (this is the same as the last option in the bottom chart's context menu)

Memory Events Filter

This window provides event filtering controls, which let you reduce the number of events displayed in the two charts so you can spot problems more easily. The window contains these dropdown panels:

Memory Events Filter

Defines memory block properties required for a memory event to appear in the charts:

Hide matching allocation/deallocation pair

Whether the block must be associated with an unmatched allocation or deallocation operation. Unmatched operations often indicate leaks or errors.

Show only events for retained objects

Whether the block must have been kept in memory instead of being deallocated. Blocks like this are common sources of leaks or errors.

Requested size range

The amount of bytes requested. For this field and other range-based fields, you can enter two values separated by a dash (`-`) to indicate the inclusive minimum and maximum range values, or enter one value for an exact match.

Band size

The band of memory that the associated block is allocated from or freed to. Bands are groups of small, preallocated memory blocks of the same size. In this field, you must enter the exact band size (not a range).

Pointer

The pointer values (addresses). You can enter one address (`0x80832c8`), or a range of addresses (`0x80832c8-0x80832f0`).

Memory Events Kind

Specifies which types of memory events will appear in the charts. Here, "event type" means which C or C++ memory-management function or operator was called (e.g., *malloc()*, *realloc()*, *new*, *delete*).

Range

Defines the range of events to display, based on either timestamps or event IDs

Files

Lists the files in which a memory event must have occurred for it to appear in the charts

Binaries and Libraries

Lists any of the executable binaries and shared library files in which a memory event must have occurred for it to appear in the charts

Threads

Lists the threads in which a memory event must have occurred for it to appear in the charts

Settings tab

This last tab lets you adjust Memory Analysis settings only for active sessions. You can't change the **Advanced Settings** but you can change any field in the **Memory Errors**, **Memory Tracing**, and **Memory Snapshots** dropdowns. Any new settings take effect when you click the **Apply** button in the toolbar.

The toolbar also has Reload and Prevent Auto-refresh buttons, similar to the toolbars in other tabs, and three buttons for performing specific operations on demand:

- Collect Memory Leaks ()
- Get Memory Snapshot ()
- Gather Allocation Traces ()

Running Memory Analysis and the GDB Debugger concurrently

If you want to debug a project while running Memory Analysis to see debugging information and memory events at the same time, you must manually configure GDB to support this workflow. To run Memory Analysis with a debugging session:

1. After selecting the project, target, and Debug mode in the launch bar, click the Edit button () on the right of the Launch Configuration dropdown.
2. Ensure that the **Stop on startup at** box is checked in the **Debug** tab, or that you have an active breakpoint at the first line in your program.
3. In the **Tools** tab, ensure that the Memory Analysis radio button is selected and the **Create control thread** box is unchecked (disabled) under Advanced Settings.

Running a separate data-collection thread can cause the application to deadlock when the debugger is also running. If you disable this thread, your program must call the **librcheck** API to perform leak checks and memory tracing at runtime. Note that you can't send signals to a target process to control **librcheck** when no control thread is running.

4. **Optional:** You can specify nondefault Memory Analysis settings. To do so:
   a. Click the **Memory** tab on the right.
   b. In the Memory Analysis controls, change any settings to customize what gets reported.

      You can adjust how Memory Analysis measures heap usage, enable or disable specific memory checks, and turn tracing on or off and specify block size tracing limits.
   c. Click **OK** to save the configuration changes and close the window.
5. When you're ready to debug and analyze your application, click the Debug button ( ).

   The IDE launches the application and attaches the debugger, which stops execution at the startup location specified in the **Debug** tab. A new Memory Analysis session is shown in **Analysis Sessions**.
6. If you don't see the `gdb` output in the **Console** view, click the **Display Selected Console** button ( ) in the upper right area and select the `gdb` entry.
7. Tell the debugger to ignore the SIGSEGV signal, with this command: `handle SIGSEGV nopass`

   Sometimes, the debugger tooling does unsafe operations and causes the OS to emit this particular signal. This directive tells `gdb` to not pass this signal onto the program.
8. Click **Resume** in the **Debug** view toolbar, or type `continue` and press **Enter** in the **Console** view.

The program continues execution until the next breakpoint. When execution stops again, you can view the stack trace and other useful information in the Debug perspective. You can also switch to the QNX Analysis perspective and view the Memory Analysis data gathered up till that point, in the editor pane.

CSV file contents for exported event data

If you export Memory Analysis results to a CSV file, the resulting data in the file depends on the event type selected in the **Export Memory Analysis Data** window.

The tables shown below list the fields exported to the CSV file, in the order in which they're written, for each event type reported by Memory Analysis. If you want to see field names in the column headers when viewing the data in a spreadsheet application (e.g., Excel), be sure to check **Generate header row** in the export controls window.

Table 1. Fields for all event types

Name	Description
`Session Name`	Abbreviated session name, which is just the binary name without the session number or timestamp shown in the **Analysis Sessions** entry.
`Session Time`	When the session was created. For an imported session, it's the time of the import, not the time of creation.
`Event ID`	Unique ID for the memory event.
`Time Stamp`	Timestamp of when the event occurred on the target machine.
`Process ID`	PID of the process.

Table 2. Additional fields for memory events

Name	Description
`Thread ID`	TID of the thread.
`CPU`	CPU number (for multicore machines).
`Alloc Kind`	Memory operation type (e.g., `malloc`, `free`) .
`Actual Size`	Number of bytes in the allocated block.
`Requested Size`	Number of bytes requested by the program.
`Deallocation`	Whether the memory block was freed.

Name	Description
`Pointer`	Pointer value associated with the event.
`Source Location`	Location in the source code where the memory was allocated.
`Root Location`	Location in the source code for the root of the stack trace; typically, this is *main()* or a thread entry function.
`Full Trace`	Full stack trace for the allocation.

Table 3. Additional fields for runtime errors

Name	Description
`Thread ID`	TID of the thread.
`CPU`	CPU number (for multicore machines).
`Message`	Error message returned.
`Pointer`	Pointer value associated with the event.
`Trap Function`	Function where the error was caught.
`Alloc Kind`	Type of allocation for the argument (pointer) being validated.
`Severity`	Error severity.
`Memory State`	Whether the pointer memory is used or free.
`Source Location`	Location in the source code where the memory was allocated.
`Root Location`	Location in the source code for the root of the stack trace; typically, this is *main()* or a thread entry function.
`Full Trace`	Full stack trace for the error.
`Full Alloc Trace`	Full allocation stack trace for the pointer.

Table 4. Additional fields for bin events

Name	Description
`Size`	Maximum size, in bytes, for the bin to which the memory belongs. The bin size ranges are based on the **Bins counters** field under **Memory Snapshots**. The default size ranges are based on powers of two, with a special bin for large allocations between 4 KB and 2 GB.
`Allocations`	Number of allocated blocks in this bin.
`Deallocations`	Number of freed blocks in this bin.

Table 5. Additional fields for band events

Name	Description
`Size`	Size, in bytes, of the preallocated band of memory used.
`Total Blocks`	Number of total blocks in this band.
`Free Blocks`	Number of free blocks in this band.

Page updated: August 11, 2025

Finding memory corruption with Valgrind Memcheck

QNX Tool Suite | Integrated Development Environment User's Guide | Developer | Setup

Valgrind Memcheck detects many types of memory corruption errors, including invalid accesses, uses of undefined values, and incorrect freeing of memory. The IDE parses the Memcheck results that it receives from Valgrind and displays a list of memory errors.

NOTE:
All Valgrind tools can be loaded and run from the command line. However, using the IDE is more convenient because it automates much of the setup by setting Valgrind command options based on UI fields and by copying the analysis results into the host workspace.

If your target image isn't configured to include the `valgrind` binaries and libraries, the IDE attempts to upload these components when you launch an application with a Valgrind tool enabled. For this to work, the target must have a writable filesystem.

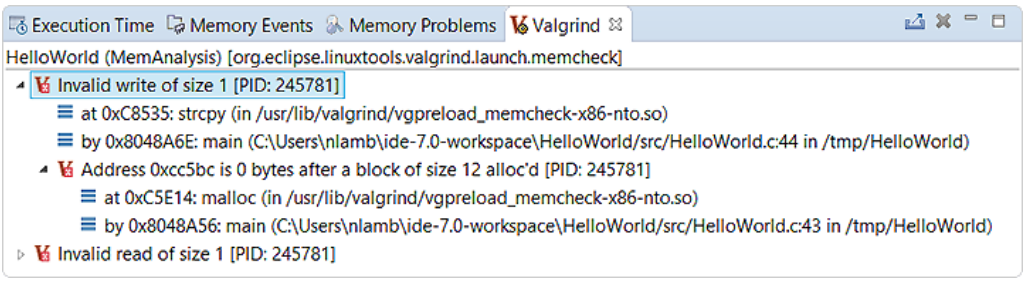
To find memory corruption with Valgrind Memcheck:

1. In the launch bar, expand the Launch Configuration dropdown (which is in the middle) and select the project for which you want to check memory.
2. In the Launch Target dropdown (on the right), select the target for running your application.
3. In the Launch Mode dropdown (on the left), select Check.
4. Click the Edit button (⚙) on the right of the Launch Configuration dropdown.
5. In the configuration editor window, access the Valgrind controls by clicking the **Check** tab on the right and then the Valgrind radio button near the top of this tab.
6. Select **Memcheck** from the **Tool to run** dropdown.
7. **Optional:** You can change any settings to customize what gets reported in the Valgrind results. The **Memcheck** tab lets you configure how the heap block data are presented.
8. Click **OK** to save the configuration changes and close the window.
9. In the launch bar, click the Check button ( ).

The IDE switches to the QNX Analysis perspective. If necessary, the IDE first builds the binary before uploading it to the target. To analyze the application, the IDE instructs Valgrind to execute the uploaded binary with Memcheck instrumentation. Then, it creates a new session for storing the Valgrind results; this session is displayed in the **Analysis Sessions** view. When the program terminates, Valgrind writes the results to a log file, which the IDE copies into the directory for the new session.

The memory error details are listed in the **Valgrind** view. All error summaries contain the Valgrind icon (V) followed by a descriptive message and the PID of the Valgrind process. On the left of the icon, you can click the arrow to display a stack trace of where the error was detected. The depth of the displayed call chain is determined by the **Callers in stack trace** field in the **General Options** tab.

If **Track origins of uninitialized values** is checked in the **Memcheck** tab, any invalid read or write error includes a stack trace of where the memory was allocated:



The stack traces should include location information for errors detected within functions of shared libraries. If you don't see this information, you must manually configure the loading of debug symbols.

If you double-click a trace line that has source file information, the IDE opens the file at the indicated line. This feature lets you quickly find where an error occured or the associated memory was allocated.

NOTE:

You can run multiple Valgrind sessions concurrently, using the same tool or different tools, on the same application or different applications. Valgrind log files always contain the PIDs of the Valgrind processes, so their names are always distinct.

Page updated: August 11, 2025

Finding memory leaks with Valgrind Memcheck

Valgrind Memcheck tracks all allocated heap blocks so it can find memory leaks when the program terminates. The tool writes the leak details into the analysis results, which the IDE then parses to display leak information.

NOTE:
All Valgrind tools can be loaded and run from the command line. However, using the IDE is more convenient because it automates much of the setup by setting Valgrind command options based on UI fields and by copying the analysis results into the host workspace.

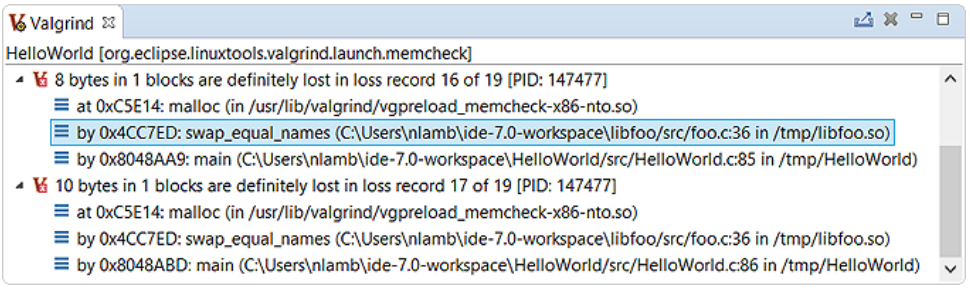
To find memory leaks with Valgrind Memcheck:

1. In the launch bar, expand the Launch Configuration dropdown (which is in the middle) and select the project in which you want to check for leaks.
2. In the Launch Target dropdown (on the right), select the target for running your application.
3. In the Launch Mode dropdown (on the left), select Check.
4. Click the Edit button ( ) on the right of the Launch Configuration dropdown.
5. In the configuration editor window, access the Valgrind controls by clicking the **Check** tab on the right and then the Valgrind radio button near the top of this tab.
6. Select **Memcheck** from the **Tool to run** dropdown.
7. **Optional:** You can change any settings to customize what gets reported in the Valgrind results.
   The **Memcheck** tab lets you enable leak checking and the reporting of individual leak types (e.g., definite, possible).
8. Click **OK** to save the configuration changes and close the window.
9. In the launch bar, click the Check button ( ).

The IDE switches to the QNX Analysis perspective. If necessary, the IDE first builds the binary before uploading it to the target. To analyze the application, the IDE instructs Valgrind to execute the uploaded binary with Memcheck instrumentation. Then, it creates a new session for storing the Valgrind results; this session is displayed in the **Analysis Sessions** view. When the program terminates, Valgrind writes the results to a log file, which the IDE copies into the directory for the new session.

The memory error details are listed in the **Valgrind** view. All error summaries contain the Valgrind icon ( ) followed by a descriptive message and the PID of the Valgrind process. On the left of the icon, you can click the arrow to display a stack trace of where the error was detected. The depth of the displayed call chain is determined by the **Callers in stack trace** field in the **General Options** tab.

For leaks, the error summary states the number of lost bytes and blocks as well as the leak type. The stack trace shows where the memory was allocated:



The stack traces should include location information for lost blocks allocated within functions of shared libraries. If you don't see this information, you must manually configure the loading of debug symbols.

If you double-click a trace line that has source file information, the IDE opens the source file at the indicated line. This feature lets you quickly find where a lost block was allocated.

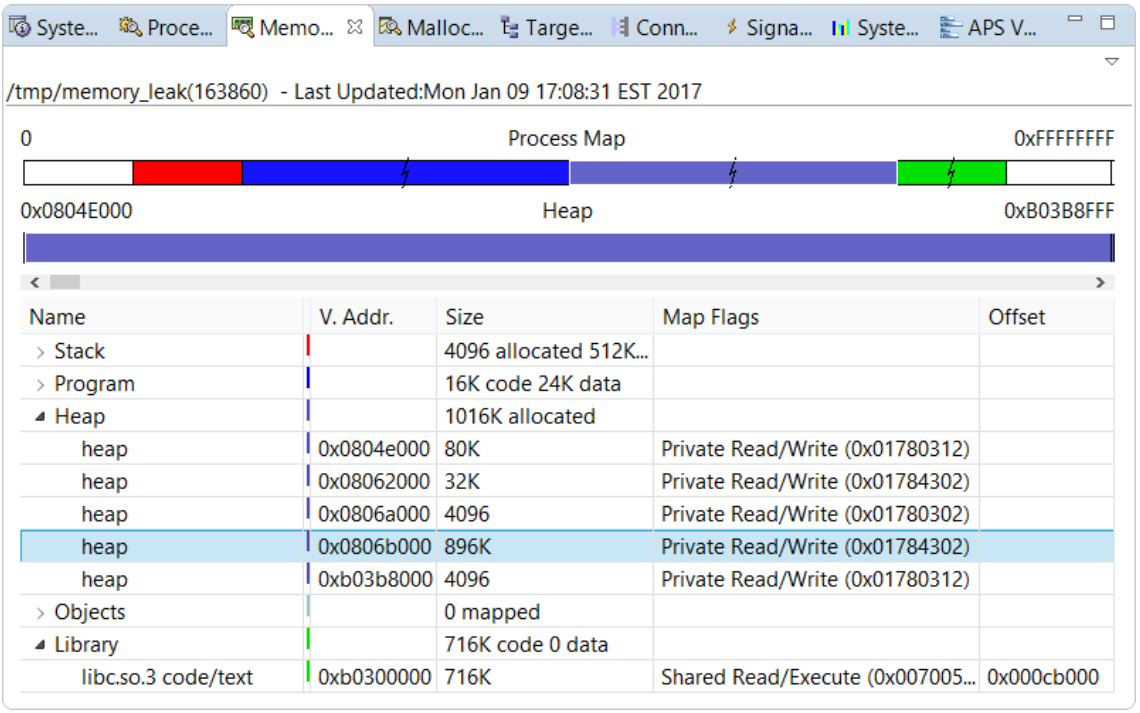
NOTE:
You can run multiple Valgrind sessions concurrently, using the same tool or different tools, on the same application or different applications. Valgrind log files always contain the PIDs of the Valgrind processes, so their names are always distinct.

Page updated: August 11, 2025

Memory Information

The **Memory Information** view gives an overview about the memory allocation pattern for a process. Specifically, it displays a memory distribution map and information about each segment.

Leaving this view open lets you monitor a process's memory usage, even when running the debugger. It's also good for spotting excessive memory usage so you can begin optimizing the heap or the stack.



At the top, the **Process Map** bar illustrates the breakdown of memory by type. The type reflects the memory's purpose (e.g., code, static data, or shared data). This bar graph is scaled so that large areas don't visually overwhelm small areas, with large areas compressed and marked with a jagged line. Meanwhile, the numbers in the top left and right corners represent the process's start and end addresses in virtual memory, which are always 0 and 0xFFFFFFFF. The memory types are color-coded as follows:

- Stack (red) — light red for the guard page, medium red for unallocated memory, dark red for allocated memory
- Program (blue) — dark blue for code (text), light blue for data
- Heap (purple) — purple for all regions, including guard pages
- Objects (cyan) — cyan for shared heap objects with certain memory flags
- Library (green) — dark green for code, light green for data

Just below, another bar shows the breakdown of the segment selected in the table further below. This second bar graph also uses color shade variations for different segment regions (e.g., guard pages versus allocated memory). Note that it's *not* scaled, meaning no compression of large areas is done. To navigate the entire bar display, use the scrollbar underneath. Here, the numbers in the corners indicate the virtual address range of the segment.

The main portion of the view contains a table with details about each memory segment. The following details are given:

Name
: Descriptive name of the segment

Virtual Address
: Virtual start address of the segment

Size
: Size of the segment. For major categories, this column lists the totals for the minor categories.

Map Flags
: Flags and protection bits enabled for the segment. For more information, see the *flags* and *prot* arguments for the *mmap()* function.

Offset
: The segment's offset into shared memory, which is equal to the *off* argument for *mmap()*.

When you click a table row, the corresponding segment is outlined in both bar graphs at the top. By default, the segments are categorized by type, but you can display them in a flat list by deselecting **Categorize** in the dropdown in the upper right corner of the view. There's also an option for copying the table contents to the clipboard, so you can take a snapshot of memory distribution details.

How memory types relate to virtual memory categories

The relation of memory types to virtual memory (VM) categories is as follows:

VM category	Memory type	Notes
Code	Program	
Shared Code	Library	Each shared library has its own code segment (and hence, table entry).
Data	Program	
Stack	Stack	Each thread in the application and shared library code has its own stack.
Heap	Heap	All heap segments for the application and shared libraries.
Shared Heap	Objects	

Page updated: August 11, 2025

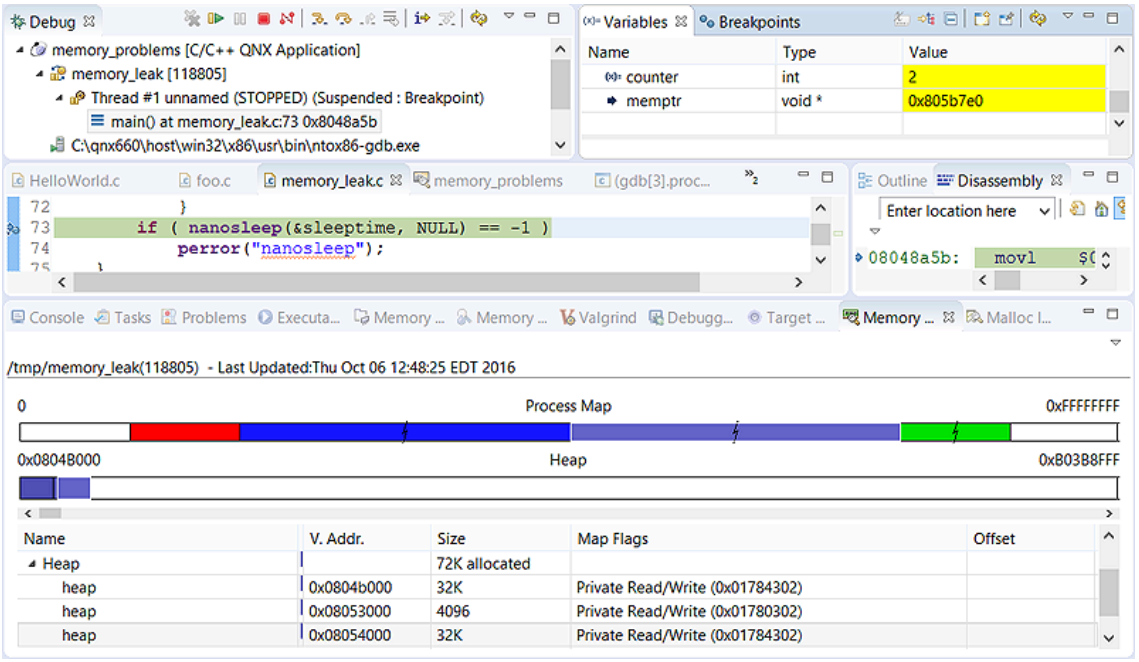
Analyzing heap memory usage with System Information and the debugger

QNX Tool Suite | Integrated Development Environment User's Guide | Developer | Setup

The QNX System Information perspective displays details about a process's memory, including heap segments. These details let you observe how the heap usage changes as you interact with an application or you move between breakpoints with the debugger.

When an application is running, you can select one of its processes in the **Target Navigator** and examine that process's memory distribution in the **Memory Information** view. If you want to know which areas of code allocate a lot of heap memory, you can add breakpoints at strategic places (see the “Using breakpoints, watchpoints, and breakpoint actions” entry in the *C/C++ Development User Guide* for information on doing so), then run the application with the debugger.

For convenience, you can open the **Memory Information** view in the Debug perspective. Then, you can start, pause, and resume execution and check how much memory is allocated between breakpoints:



Page updated: August 11, 2025

Monitoring memory consumption at the process level

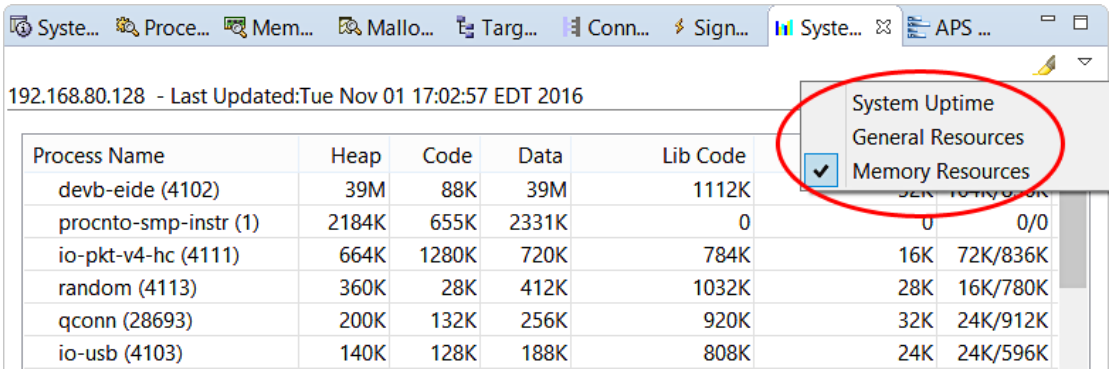
QNX Tool Suite | Integrated Development Environment User's Guide | Developer | Setup

The QNX System Information perspective contains multiple views that show memory usage data about target processes. These data let you compare the memory footprint of different processes, see the distribution of memory in different segments, and observe changes in a process's heap usage.

These activities are often the first steps in optimizing an application's memory usage because the data values shown in the IDE tell you which segments (e.g., stack, heap) are too big or growing too quickly.

Comparing memory segment sizes of target processes

When a process is consuming excessive memory, it's usually because the heap is growing too much. After selecting a target in the **Target Navigator**, you can access the **System Resources** view and select **Memory Resources** in the upper right dropdown to see the heap sizes of the target processes:

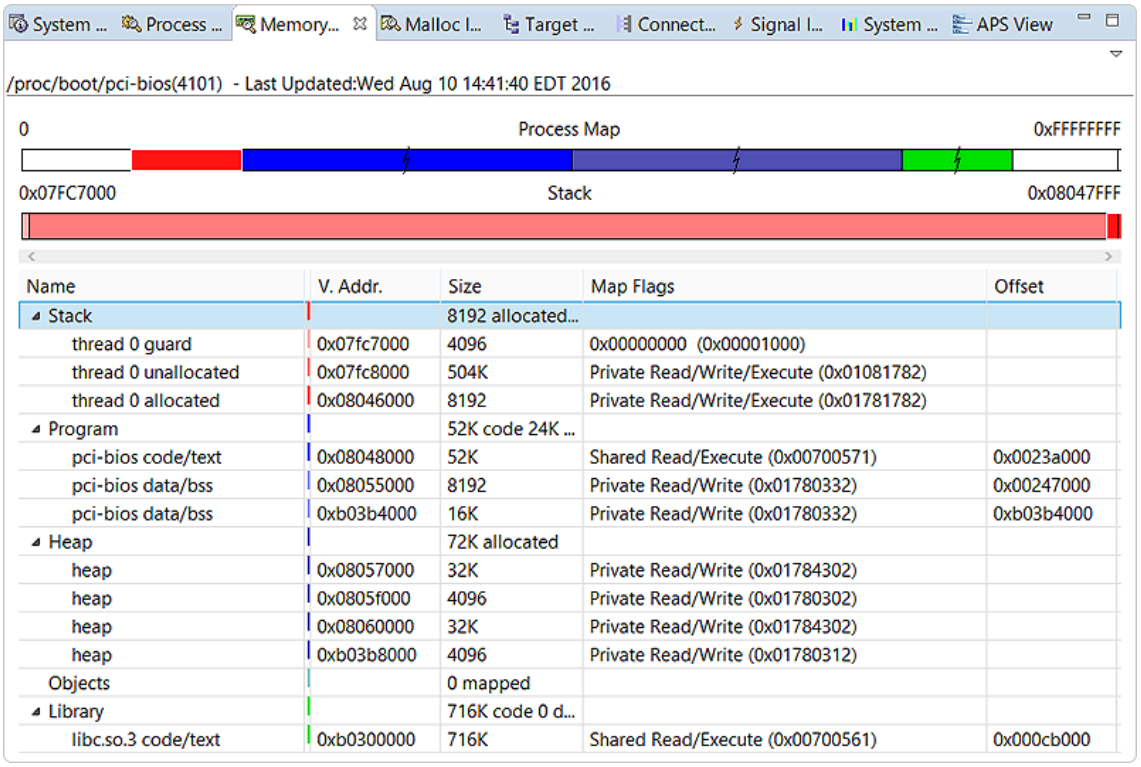


Process Name	Heap	Code	Data	Lib Code		
devb-eide (4102)	39M	88K	39M	1112K		
procnto-smp-instr (1)	2184K	655K	2331K	0	0	0/0
io-pkt-v4-hc (4111)	664K	1280K	720K	784K	16K	72K/836K
random (4113)	360K	28K	412K	1032K	28K	16K/780K
qconn (28693)	200K	132K	256K	920K	32K	24K/912K
io-usb (4103)	140K	128K	188K	808K	24K	24K/596K

The view then displays the heap, code, and data segment size for each process. You can sort the process list based on any metric, by clicking the corresponding column header, and highlight any changed values by clicking the highlight button in the upper right toolbar ( ); further details are given in the *System Resources reference*.

Examining memory distribution for a process

The **Memory Information** view displays details about all memory types for the process selected in the **Target Navigator**. This information includes a memory distribution map and a table showing the names, sizes, and other details about the various segments:



Name	V. Addr.	Size	Map Flags	Offset
Stack		8192 allocated...		
thread 0 guard	0x07fc7000	4096	0x00000000 (0x00001000)	
thread 0 unallocated	0x07fc8000	504K	Private Read/Write/Execute (0x01081782)	
thread 0 allocated	0x08046000	8192	Private Read/Write/Execute (0x01781782)	
Program		52K code 24K ...		
pci-bios code/text	0x08048000	52K	Shared Read/Execute (0x00700571)	0x0023a000
pci-bios data/bss	0x08055000	8192	Private Read/Write (0x01780332)	0x00247000
pci-bios data/bss	0xb03b4000	16K	Private Read/Write (0x01780332)	0xb03b4000
Heap		72K allocated		
heap	0x08057000	32K	Private Read/Write (0x01784302)	
heap	0x0805f000	4096	Private Read/Write (0x01780302)	
heap	0x08060000	32K	Private Read/Write (0x01784302)	
heap	0xb03b8000	4096	Private Read/Write (0x01780312)	
Objects		0 mapped		
Library		716K code 0 d...		
libc.so.3 code/text	0xb0300000	716K	Shared Read/Execute (0x00700561)	0x000cb000

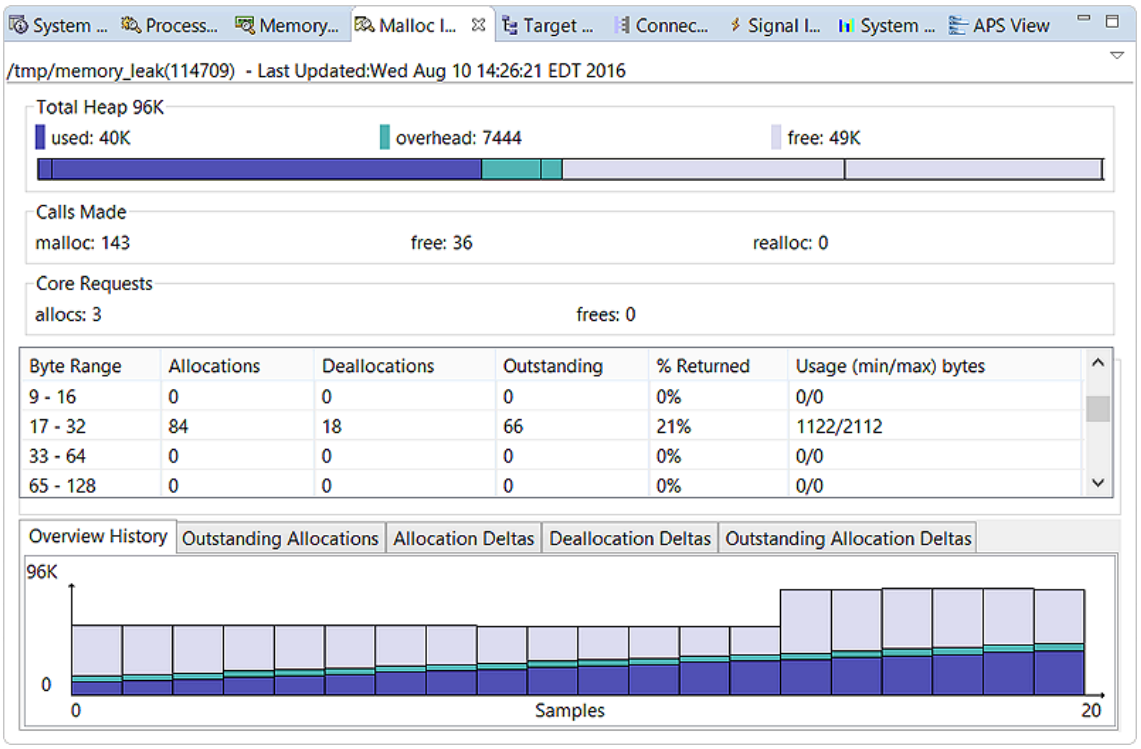
The **Process Map** bar at the top shows the breakdown of process memory based on type (e.g., Stack, Program, Heap). The memory types are color-coded, with large areas compressed and marked with a jagged line. The numbers in the top left and right corners represent the process's start and end addresses in virtual memory.

Just below is another bar, which shows the breakdown of the memory segment selected in the table further below. Initially, the segment representing the entire stack is selected. Here, the numbers in the corners indicate the virtual address range of the segment.

The table lists the name, virtual address, size, map flags, and offset (when applicable) for each memory segment. Details about these fields are given in the Memory Information reference. The table selection is reflected in both bar graphs at the top.

Monitoring heap memory consumption

The **Malloc Information** view lets you monitor a process's managed heap in real time. If you click a process entry in the **Target Navigator**, the **Malloc Information** view then displays the process's total allocator managed memory and allocation history:



NOTE:
The screenshot above represents a system targeting SDP 7.1. In an SDP 8.0 target, there's less data available.

CAUTION:
If you're using the debug allocation library, **librcheck**, the data displayed in this view might not be accurate. In this case, use Memory Analysis to view heap usage.

In the table that lists allocation counts, you can watch the **Outstanding** column to see if the program is allocating memory in a certain size range faster than freeing it, which can degrade performance. To observe changes in the overall heap usage, you can examine the **Overview History** graph. In this example, the program's used section of the heap is growing steadily, causing the overall heap size to increase as well. You would then want to look for memory leaks.

Full details on all statistics shown in this view are given in the Malloc Information reference.

Page updated: August 11, 2025

Monitoring memory and resource consumption

The QNX System Information perspective displays realtime data about the processes running on a target machine, including memory and resource usage statistics.

This perspective is a good starting point for learning about application memory usage because its views let you see data about each application's memory layout and usage. Furthermore, the data values come in real time from the target (through **qconn**), without you having to recompile or relaunch an application.

NOTE:
To see data from a particular target in the QNX System Information perspective, you must select that target in the **Target Navigator**. In this perspective, the data values in the views are refreshed every five seconds, allowing you to closely monitor the system's or a process's memory or resource consumption.

First, you can use the **System Summary** view to see the target's memory consumption data, which include how much memory the processes are using relative to each other. Then, you can access the **Memory Information** and **Malloc Information** views to monitor process memory consumption.

Finally, the **Connection Information** view lets you monitor the resources used in a process's connections.

QNX OS utilities for monitoring memory consumption

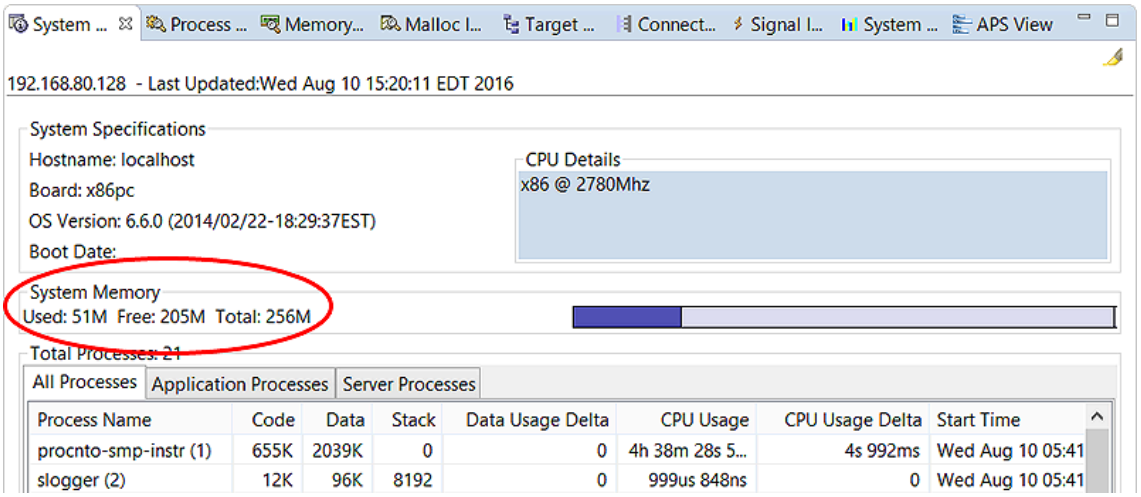
The QNX OS includes some useful memory-monitoring utilities. One of them is `pidin`, which shows process memory details, such as the current and maximum stack memory per thread.

Page updated: August 11, 2025

Monitoring memory consumption at the system level

The **System Summary** view shows the total memory usage of the target selected in the **Target Navigator** and lists the running processes and some of their memory segment sizes. This information lets you recognize when a target is low on memory and see which processes are using the most memory.

By glancing at the **System Memory** pane, you can see right away if your target is running low on RAM:



Below this area, the **Total Processes** pane lists the running processes and their code, data, and stack segment sizes along with some statistics on their CPU usage. This list tells you how much memory the processes are using relative to each other, which helps you identify which ones you might want to monitor individually.

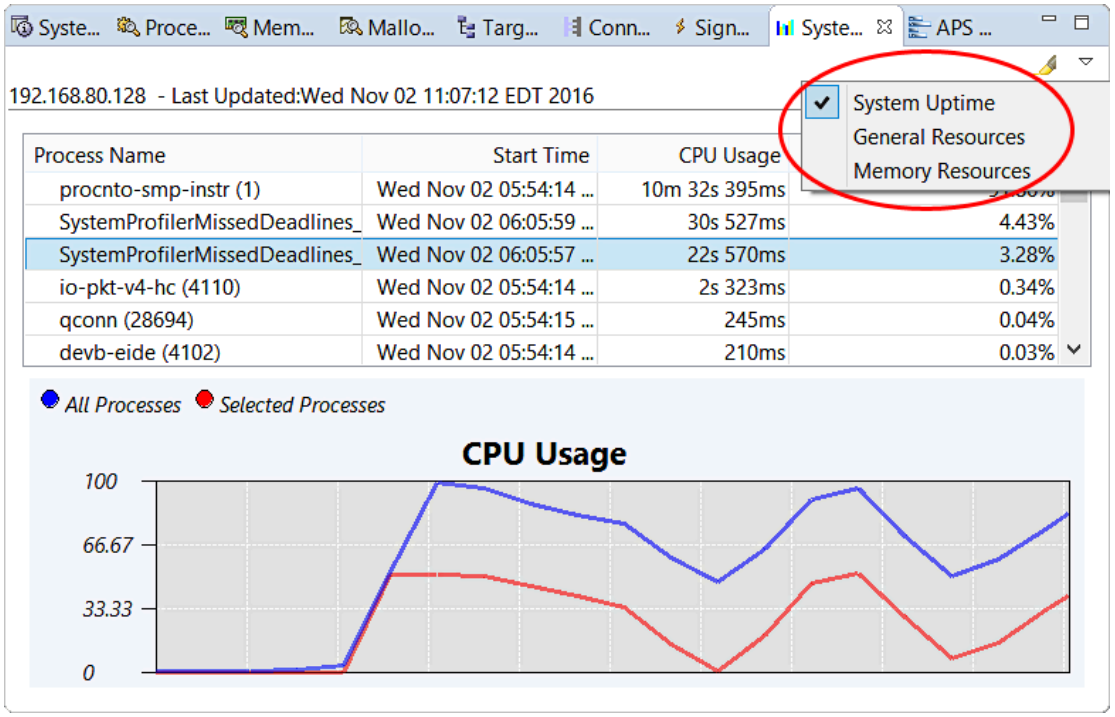
Page updated: August 11, 2025

Monitoring performance of processes

The QNX System Information perspective contains views that let you compare the CPU usage of different processes and see how their usage levels are changing.

Comparing CPU usage of processes

After selecting a target in the **Target Navigator**, you can see the CPU usage of its processes by accessing the **System Resources** view, then clicking the dropdown button (▽) in the upper right corner and selecting **System Uptime**:

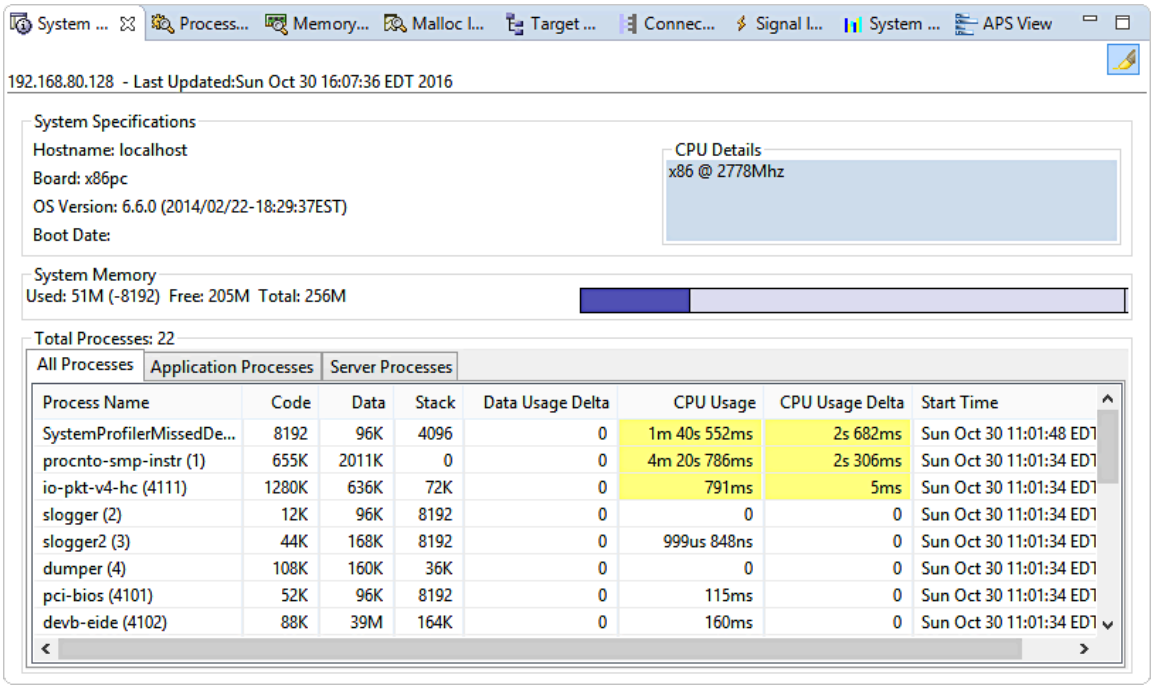


The view then displays a table with the start time, total CPU usage time, and usage as a percentage of system uptime, for each process. You can sort the table based on any displayed metric, by clicking the corresponding column header. To easily spot changes in any metric, click the highlight button (🖉) in the upper right toolbar. The view then colors in the cells containing any values that changed since the last update. You can spot a process that's consuming many CPU cycles at the present moment by its steadily increasing Uptime value (shown in the rightmost column).

Below the table, the **CPU Usage** graph shows the percentage of cycles consumed by various processes over time. By default, the **All Processes** line is enabled, so the graph illustrates the total CPU usage of all processes. You can also display the cycles used by specific processes, by clicking their names in the table and ensuring that **Selected Processes** is enabled. In this example, a single process is selected and the graph shows that its CPU usage is affecting the overall system load.

Examining changes in process CPU usage

If after viewing the **CPU Usage** graph, you suspect that certain processes are imposing a heavy computational load, you can examine their CPU usage changes through the **System Summary** view:



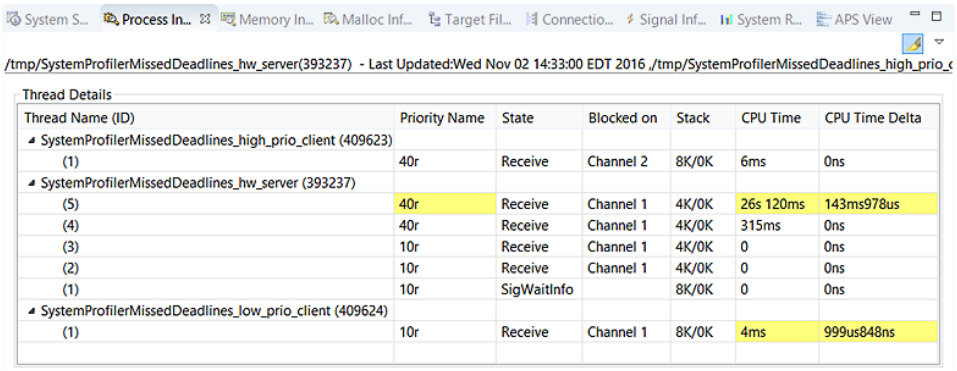
Here, the highlight button has been clicked so that the cells with changed values are colored in, and the process list has been sorted by the **CPU Usage Delta** column. This metric measures the increase in cumulative CPU time for a process since the last update of the statistics. The default update rate is 5 seconds, so we can see that the `SystemProfilerMissedDeadlines` process is currently using many CPU cycles because its **CPU Usage Delta** value is significant compared to that rate.

Page updated: August 11, 2025

Monitoring performance of threads

The **Process Information** view displays performance data for threads belonging to the processes selected in the **Target Navigator**. These data include CPU times, so you can see which threads are consuming the most processor cycles.

In the **Thread Details** pane, the statistics are given in a table:



Thread Name (ID)	Priority Name	State	Blocked on	Stack	CPU Time	CPU Time Delta
SystemProfilerMissedDeadlines_high_prio_client (409623)						
(1)	40r	Receive	Channel 2	8K/0K	6ms	0ns
SystemProfilerMissedDeadlines_hw_server (393237)						
(5)	40r	Receive	Channel 1	4K/0K	26s 120ms	143ms978us
(4)	40r	Receive	Channel 1	4K/0K	315ms	0ns
(3)	10r	Receive	Channel 1	4K/0K	0	0ns
(2)	10r	Receive	Channel 1	4K/0K	0	0ns
(1)	10r	SigWaitInfo		8K/0K	0	0ns
SystemProfilerMissedDeadlines_low_prio_client (409624)						
(1)	10r	Receive	Channel 1	8K/0K	4ms	999us848ns

This example shows statistics from three processes—a multi-threaded server and two single-threaded clients. The highlight option ( ) is enabled, so we can easily spot changed values. In this case, thread number 5 is currently the busiest server thread, as seen by its changed CPU time values. Meanwhile, the lower priority client has been the more active of the clients since the last update because its CPU time values have also changed.

You can adjust which statistics are displayed through the dropdown control ( ) in the upper right corner; further details are given in the *Process Information reference*.

Page updated: August 11, 2025

Monitoring performance

The QNX System Information perspective displays realtime data about the processes running on a target machine, including CPU usage statistics. These statistics help you find processes or threads that are consuming many CPU cycles.

This perspective is a good starting point for assessing an application's impact on system responsiveness because its views let you easily compare the CPU usage of different processes. Furthermore, the data come in real time from the target (through **qconn**), without you having to recompile or relaunch an application.

NOTE:
To see data from a particular target in the QNX System Information perspective, you must select that target in the **Target Navigator**. In all perspective views, the data are refreshed every five seconds, allowing you to closely monitor performance for the entire system or individual processes.

First, you can use the **System Resources** view to see the current and past computational load on the target machine. The **System Summary** view displays similar statistics but includes the change in CPU usage for each process. Then, you can switch to the **Process Information** view to examine thread-level CPU times for selected processes.

QNX OS utilities for monitoring performance

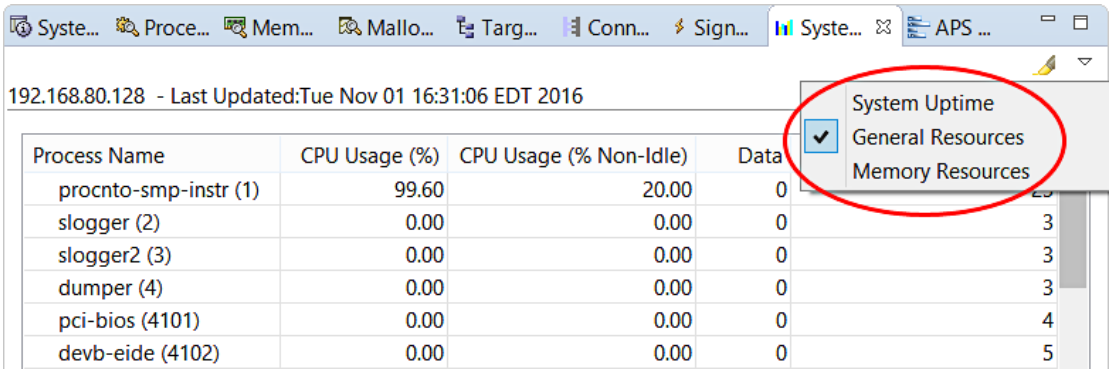
The QNX OS includes several utilities that monitor process performance. You can run the `hogs` command to list the processes hogging the CPU. The `top` utility provides more details about process performance, including thread states and CPU usage. Also, run `pidin` lets you view performance metrics for processes, such as the CPU seconds consumed by the kernel on behalf of a process or by its children.

Page updated: August 11, 2025

Monitoring resource usage for a process

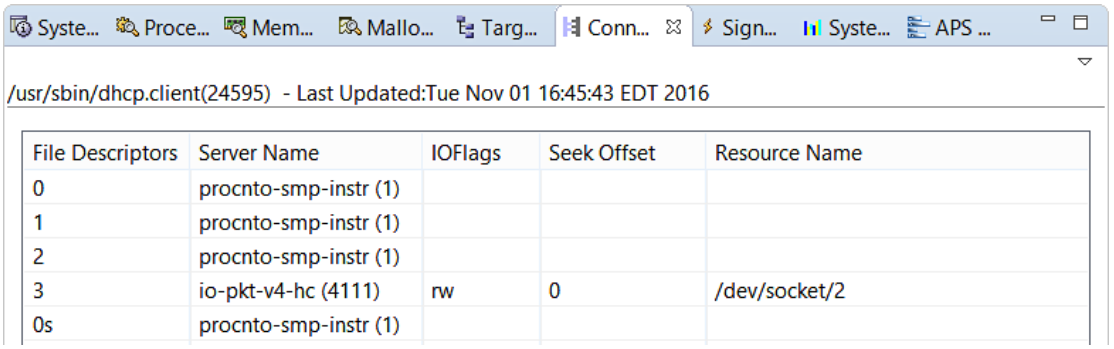
You can monitor the resource usage for a process through two QNX System Information views that list the file descriptors and other details about the process's active connections.

To see some of the resources used by each target process, you can access the **System Resources** view and select **General Resources** in the upper right dropdown. The view then lists the number of open file descriptors along with CPU usage and data segment size, for all processes:



You can examine the **File Descriptors** column (on the far right) to see if the number is steadily increasing for a particular process. If so, that process is opening file descriptors faster than closing them and might have a design or coding bug. Having too many open file descriptors can notably increase a process's memory footprint or cause it to run out of descriptors and stop working. You can click the column header to sort the process list by this metric, to find which processes are using the most descriptors.

To find out what they're used for, you must access the **Connection Information** view:



The connections are sorted by the **File Descriptors** field, in ascending order. For details about the file descriptor numbers (including the s next to some of them) and other fields, see the *Connection Information reference*. In summary, **Server Name** specifies the process at the other end of the connection, while **Resource Name** provides the device path (when applicable). These fields can give you a good idea of which program areas might be leaking descriptors.

Page updated: August 11, 2025

Optimizing an application after analysis

The memory-analyzing tools tell you the total memory a process is using, the sizes of its memory segments, and the history and breakdown of its heap usage. This knowledge helps you determine what programming steps are needed to reduce an application's memory footprint, which can improve performance.

Memory efficiency is often critical in embedded systems, where memory is limited (especially with the absence of swapping) and many processes need to run continuously. The optimization steps you'll want to take depend on what the analysis results reveal about memory type distribution. For example, you can spend considerable time optimizing the heap but if your program uses more static memory than it should, this other problem must be dealt with.

Memory distribution of processes

Virtual memory occupied by a process is separated into these categories:

- Code — Executable code (instructions) belonging to the application or static libraries.
- Shared Code — Executable code from shared libraries. If many processes use the same library, their virtual segments containing its code are mapped to the same physical segment.
- Data — A data segment for the application and data segments for the shared libraries. This memory type is usually referred to as *static memory*.
- Stack — Memory required for function stacks (there's one stack per thread).
- Heap — All memory dynamically allocated by the process.
- Shared Heap — Other memory allocated by different means, including shared and mapped memory.

The IDE has several tools for viewing process memory distribution. In the System Information, the **Memory Information** view shows the memory breakdown by type and provides details about individual segments. Note that "type" is different from virtual memory category; the correspondance is given in "How memory types relate to virtual memory categories".

You can view the heap distribution through the **Malloc Information** view, which displays the used, overhead, and free heap memory sizes. The Memory Analysis tool graphs this same information as well as all heap allocations and deallocations, in an interactive editor window. Through the Valgrind UI controls, you can run Massif to collect heap snapshots, then analyze the heap breakdown measured at the detailed snapshots.

After examining the memory distribution data with these tools, you should focus on the areas of high consumption for nonshared memory. Note that "nonshared memory" can include stack and heap memory used by shared libraries. This term covers anything *not* created as a shared memory object; this last concept is explained in the "Shared memory" entry of the *System Architecture* guide. Optimizing shared memory is unlikely to notably reduce the overall memory consumption on the target machine.

The techniques for improving memory efficiency greatly vary for different memory types. We outline some of these techniques below.

Heap optimizations

You can use the following techniques to optimize the heap:

Eliminate explicit memory leaks

The easiest way to begin optimizing the heap is to eliminate explicit memory leaks, which occur when blocks become inaccessible because their pointer values aren't kept properly. Memory Analysis lets you check for leaks at fixed intervals and outputs a list of memory errors and tags any leaks with a keyword. Valgrind Memcheck can check for specific leak types, to identify leaks resulting from incorrect pointer values or broken pointer chains.

Eliminate implicit memory leaks

After fixing the explicit leaks, you should fix the implicit leaks. These are leaks caused by heap objects that keep growing in size but remain accessible through pointers. To find such cases, Memory Analysis lets you filter the results to see only events for unmatched allocations or deallocations or for blocks that remain in memory for the program's duration. Viewing these events lets you find places where the program is steadily accumulating memory.

Valgrind Massif gathers heap data that reveal the change in heap breakdown over time, which helps you spot increasing memory usage at precise locations. Note that the Valgrind User Manual refers to these situations

as *space leaks*.

Reduce heap fragmentation

Heap fragmentation occurs when a process accumulates many free blocks of varying size in noncontiguous addresses. In this case, the process will often allocate another physical page even if it seems to have enough free memory.

The QNX OS memory allocator already solves most of this problem by preallocating many small, fixed-size blocks known as *bands*. Using bands lets the allocator quickly find a free block that fits the request size well, thereby minimizing fragmentation.

In the Memory Analysis editor, you can inspect the heap fragmentation by reviewing the Bins or Bands graphs. An indication of serious fragmentation is if the number of free blocks of smaller sizes grows over time. To deal with this, you can reorder heap allocations in your program. By allocating the largest blocks first, you'll reduce how often the allocator must divide large blocks into smaller ones. Whenever this happens, the smaller blocks can't be used later for bigger blocks because the address space is not contiguous.

If your program logic allows for it, you can store data in multiple smaller structures that each fit within the largest preallocated band size (typically, 128 bytes). Whenever a request exceeds this size, the block is allocated in the general heap list, which means a slower allocation and more fragmentation.

Reduce the overhead of allocated objects

There are several sources of overhead for heap-allocated objects:

- User overhead — The application might request more heap memory than it really needs. This often results from predictive algorithms, such as those used by *realloc()*. You can reduce this overhead by better estimating the average data size. To do this for a particular call chain, examine the related allocation backtraces in the **Memory Backtrace** view. Or, if your data model allows it, truncate the memory to fit into the actual size of the object, after the data growth stops.
- Padding overhead — In programs that run on processors with alignment restrictions, the fields in a `struct` type can get arranged in a way that makes the overall size of the structure larger than the sum of the sizes of its individual fields. You can save some space by rearranging the fields; usually, it's better to put fields of the same type together. You can measure the result by writing a *sizeof* test. Typically, this task is valuable when the resulting overall size matches a preallocated band size (see below).
- Block overhead — Sometimes there's extra space in heap blocks because the memory allocated is more than what's requested. In the Memory Analysis results, the **Memory Events** view shows the requested versus actual allocation sizes and the **Usage** tab shows what percentage of the heap is overhead (extra space). Whenever possible, choose an allocation size that matches a size for preallocated bands (you can see their sizes in the **Bands** tab), especially for *realloc()* calls. Also, if you can, try to align data structures with these band sizes.

Tune the allocator

Occasionally, application-driven data structures have fixed sizes and you can improve memory efficiency by customizing the allocated block sizes. Or, your application may experience *free blocks overhead*, when a lot of memory has been freed by the code but the process hasn't returned many pages. This happens if the process doesn't reach the “low watermark” on heap usage, which causes it to return some pages. In these two cases, you must either write your own allocator or contact QNX to obtain a customizable allocator.

To estimate the benefits of custom block sizes, configure Memory Analysis to report the allocation counts for the appropriate size ranges, by setting the **Bins counters** field in the Memory Snapshots controls. Then, examine the **Bins** tab in the analysis results to see the distribution of heap objects within the bins (size ranges) that you specified.

Code optimizations

In embedded systems, it's very important to optimize the size of an executable or library binary because it uses not only RAM memory but expensive flash memory. You can use the following techniques:

- Ensure that the binary file is compiled without debug information when you measure it. Debug information is the largest contributor to file size.
- Strip the binary to remove any remaining symbol information.
- Remove any unused functions.
- Find and eliminate code clones.
- Try setting compiler optimization flags (e.g., `-O`, `-O2`). Note that there is no guarantee that the code will be smaller; it can actually be larger in some cases.

- Don't use the `char` type to perform `int` arithmetics, particularly for local variables. Converting between these types requires the compiler to insert code, which affects performance and code size, especially on ARM processors.
- Bit fields are also very expensive in arithmetics on all platforms; it's better to use bit arithmetics explicitly to avoid hidden costs of conversions.

Data optimizations

Static memory can produce significant overhead, similar to heap or stack memory. You can take some steps to reduce the size of an application's data segments:

- Inspect global arrays that consume a lot of static memory. It may be better to use the heap, particularly for objects that aren't used throughout the program's entire lifetime.
- Find and remove unused global variables.
- Determine if any structures have padding overhead. If so, consider rearranging their fields to achieve a smaller overall size.

Stack optimizations

Sometimes, it's worth the effort to optimize the stack. For example, your application may have frequent high peaks in stack activity, meaning that large stack segments constantly get mapped to physical memory. These situations can be hard to detect through conventional testing. Although the program might run properly during testing, the system could fail in the field, likely when it's busiest and needed the most.

You can watch the **Memory Information** view for stack allocation statistics and then locate and fix code that uses the stack heavily. Typically, heavy stack usage occurs in two situations: recursive calls, which should be avoided in embedded systems, and usage of many large local variables, such as arrays kept on the stack.

Page updated: August 11, 2025

Preparing your target

The QNX development environment consists of a host and a target machine.

The *host* is where you write and build your code. This machine runs Windows or Linux and contains the QNX SDP and IDE. The *target* is where you run, debug, and profile your applications. This machine runs QNX OS and contains the embedded system you're designing (and all of the applications for it).

The target can be a physical board or an existing virtual machine. In these two cases, you must manually set up host-target communication, which can be based on an IP or a serial connection. As of version 7.1 of the IDE, the target can also be a QNX virtual machine (VM) that you create through the IDE, as explained in "Creating a QNX virtual machine". Creating your own VM is convenient because the generated QNX system image includes the drivers and utilities needed for host-target communcation, saving you from manually setting it up.

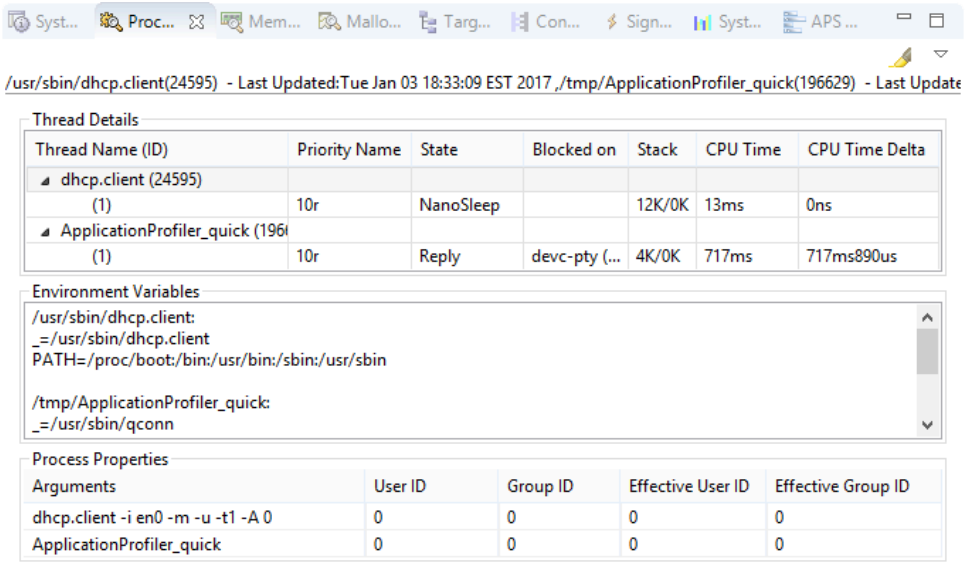
For all target types, you must create a target connection to tell the IDE where your applications will run. The next section explains how to create a connection to an existing target.

Page updated: August 11, 2025

Process Information

The **Process Information** view provides runtime details about processes and threads. For this view, information is shown for each process selected in the **Target Navigator**.

Individual panes within the view present thread state information, environment variables, and user and group IDs for processes, as described below. This view is handy for monitoring performance of threads, detecting deadlock, and verifying the proper priorities of threads.



Thread Details

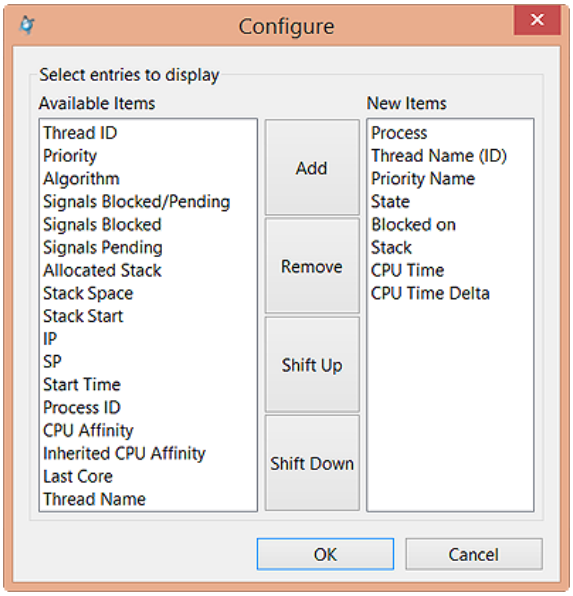
This top pane displays thread details in a table. For each process, its binary name and PID is given in one row and its threads and their properties are listed in the rows underneath. You can toggle the display of a process's threads by clicking the arrowhead next to its name. This lets you filter the table to see only some threads. If you don't want to see any process rows, click the dropdown button (▽) in the upper right corner of the view, then click **Tree View**, to deselect the grouping by process feature.

By default, the following columns are shown:

- Thread Name (ID)
- Priority Name
- State
- Blocked on
- Stack
- CPU Time
- CPU Time Delta

You can sort the thread list by any displayed metric, by clicking the corresponding column header. To easily spot changed values, click the highlight button (🖉) in the upper right toolbar of the view. The view then colors in the cells containing any values that changed since the last update. You can change the highlight color in the System Information preferences, by selecting **Window** > (and then)**Preferences** > (and then)**General** > (and then)**Appearance** > (and then)**Colors and Fonts** > (and then)**System Information** > (and then)**Highlight color**.

You can customize which columns are shown by clicking **Configure** from the dropdown menu. This action opens a popup window that lets you choose which thread details are shown and in which order they're listed in the table (from left to right):



If you right-click a process row in the table, you see menu options for interacting with processes. These are the same options shown in the **Target Navigator** menu. If you right-click a thread row, you see similar options, except that you can't slay a thread or deliver a signal to it. Also, in addition to setting the priority and the inherited and non-inherited processor affinities, you can set the thread name.

Environment Variables

The environment variable settings for each process are listed in this pane. For information about environment variables used on QNX targets, see the "Commonly Used Environment Variables" section in the *Utilities Reference*.

Process Properties

This bottom pane lists each process's command line, including arguments, and its real and effective user and group IDs.

The arguments listed are the ones used to start a process, which means they're what was passed to the executable binary, not necessarily what you might have entered on the command line. For example, if you entered `wc *.c`, the pane might show `wc cursor.c io.c my.c phditto.c swaprelay.c` because the shell expands the `*.c` token before launching the program.

The user and group IDs determine which permissions are used for a process. Suppose you start a process as **root** but use *seteuid()* and *setegid()* to run the program as the user **jsmith**. The program then runs with the permissions of **jsmith**. By default, all programs launched from the IDE run as **root**.

Page updated: August 11, 2025

Controlling profiling through API calls

In your application code, you can use special macros to turn profiling on and off and profiler API functions to change which metrics are reported. Calls to these macros and functions modify the parameters and behavior of **libprofilingS**.

Adapting your code to use this API (and rebuilding and relaunching the application) is admittedly more work than configuring Application Profiler settings in the IDE or sending signals to a target process. However, the API allows you to profile specific code regions without trying to perfectly time signal delivery, and to dynamically change the profiling method and time units used.

Here, we demonstrate how to use some common commands.

Sample program

Your program must include the profiler header file (**qprofiler.h**) and call *QPROFILER_START()* and *QPROFILER_STOP()* as needed to profile the appropriate areas. For these macros to do anything, your makefile must define the QPROFILING macro for compiling and link with the **libprofilingS** library (for details, see "Enabling function instrumentation").

Consider the following program:

```
#include <stdlib.h>
#include <stdio.h>
#include <process.h>
#include <qprofiler.h>

#define NUM_ITEMS 32767

void quickSort(int numbers[], int array_size);
void q_sort(int numbers[], int left, int right);

int numbers[NUM_ITEMS];

int main() {
  int i;
  srand(getpid());
  // fill array with random integers
  for (i = 0; i < NUM_ITEMS; i++) {
    numbers[i] = rand();
  }

  QPROFILER_START();

  // perform quick sort on array
  quickSort(numbers, NUM_ITEMS);

  QPROFILER_STOP();

  printf("Done with sort.\n");
```

This program generates an array of random numbers, uses the quick sort algorithm to sort it, then displays the sorted array. The profiling is turned on just before the sorting begins and turned off just after it ends. So if you set *QPROF_AUTO_START* to 0 (to prevent profiling from starting automatically) when you run this program, you'll see function measurements strictly for the sorting code.

Using macros to start and stop profiling is convenient because you can leave them in production code that gets built without the QPROFILING macro defined or the **libprofilingS** library being linked. In this case, the macros expand to nothing and thus, have no effect.

The profiler API also offers functions to change the metrics reported; to use these functions, you must build with the appropriate profiling options. Consider this second sample program:

```
// Parallel sorting by regular sampling --
// A four-phase parallel sorting algorithm in which each process:
// 1) Locally sorts a subset of the original list
// 2) Takes samples (element values) from its sorted list subset and
sends them to
//    the master process, which sorts them to determine pivot values
// 3) Based on pivot values received from the master, sends its list
partitions to
//    the other processes, with each partition containing values within
a certain
//    range and designated for the process responsible for sorting that
range
// 4) Merges (no resorting is necessary) the received partitions to
produce a
//    fragment of the final, sorted list
int psrs() {
    ...
    // Phase three -- exchange list partitions with other processes
    if ((parentProcess) &&
        (retval = qprofiler_set_mode(MODE_KERNEL_TRACE,
METHOD_REALTIME)) == -1) {
        // Error-handling code goes here
    }

    // Send each partition to the process in charge of sorting that
value range
    for (i = 0; i < NUM_PROCESSES; i++) {
        if (i != myProcessIndex) {
```

This program implements a multiphase parallel sorting algorithm. In the third phase, each process sends partitions of its locally sorted list subset to the other processes. Suppose you want to analyze the target system's performance during this communication-intensive phase. One process can then enable kernel tracing mode just before the partition exchanging begins, then renable function runtime measurement after it ends.

To start the kernel event trace at this exact code location, the sample program calls *qprofiler_set_mode()* with the MODE_KERNEL_TRACE flag set. Although the Application Profiler controls allow you to set a delay (in seconds) for starting a trace after the application is launched, this feature isn't precise enough for multiprocess computations where timing is unpredictable. When the third phase is completed, the program calls that function again but with the MODE_LOG_FILE flag set, to resume writing function runtimes to the output file.

The API also provides a function for obtaining call chain information:

```
qprofiler_callstack_t[MAX_BACKTRACE_DEPTH] self_backtrace;
...
if (qprofiler_backtrace_self(
        &self_backtrace, MAX_BACKTRACE_DEPTH, 0) == -1) {
    // Error-handling code goes here
}
else {
    // Code to write out or use call chain addresses goes here
}
```

The *qprofiler_backtrace_self()* function fills the provided buffer with stack frame information for the current backtrace (call chain). This is useful if you've built your binary with call count instrumentation (and hence, are profiling with the MODE_BACKTRACING setting) and you want to display call chain details, perhaps for debugging.

Page updated: August 11, 2025

Controlling profiling through signals

You can pause (temporarily stop) and resume (restart) profiling activity by sending POSIX signals to the process for a binary built with profiling instrumentation. Using signals lets you gather profiling data at specific times as an application runs.

NOTE:
For this IPC mechanism to work, you must register signal handlers before launching the application. You can do this through the **Control** fields in the Application Profiler UI or by setting the *QPROF_SIG_STOP_PROFILING* and *QPROF_SIG_CONT_PROFILING* environment variables on the command line that launches the application.

In the IDE, you can send signals to target processes by using the **Target Navigator**. You must use whatever signals are defined in the **Control** fields; by default, these signals are 17 for pausing profiling and 16 for resuming it.

On the command line, you can use the `kill` command. Here, you must use the signals defined in the environment variables. Note that in either case, you can't change the signal mapping at runtime.

Page updated: August 11, 2025

Controlling profiling at runtime

You can configure the initial profiling settings through the Application Profiler in the IDE or through command-line settings. While an instrumented binary runs, you can further control profiling by using signals or API calls.

The exact capabilities of these two runtime control methods are as follows:

Signalling

If you installed signal handlers to dynamically control profiling, whether through the Application Profiler controls or environment variables on the command line, you can send signals to the instrumented binary to pause and resume profiling.

API calls

The **libprofilingS** library defines macros for pausing and resuming profiling. Your application code can call these macros to collect profiler data in specific regions only. It can also call functions to change the profiling mode and timer resolution, and to get backtraces.

You can use the Application Profiler or a command line to set the initial profiling behavior (details on doing so are given in the next subsection) but then use one or both runtime control methods to refine the profiling activity. Any adjustments made at runtime override any equivalent environment variable settings that were in effect when the program was launched.

Defining initial settings

The initial profiling settings are defined through environment variables. When you launch an application with profiling enabled from the IDE, the Application Profiler sets these variables based on its UI fields. For Functions Instrumentation mode, you can prevent profiling from automatically starting or define signals for pausing and resuming profiling activity, by setting the **Control** fields.

When launching applications outside of the IDE, you can set the environment variables ahead of time (e.g., in a setup script) or on the command line. The minimum variable settings needed to see profiling results are given in "Running an instrumented application binary".

Page updated: August 11, 2025

QNX launch configuration types

Launch configurations are based on templates of property settings that control program deployment, execution, or analysis. Unlike launch modes, launch configuration types are designed for specific tasks, not general use cases.

Here, we explain the QNX launch configuration types for performing tasks on QNX OS systems. We list and describe the configuration properties that you can set in "QNX launch configuration properties". A QNX launch configuration doesn't have to involve running a QNX project; it can also automate a setup task, such as copying files to the target, or an analysis task, such as running a kernel event trace.

NOTE:
The **Launch Configuration Type** window shows only those launch configuration types that support the launch mode you selected in the **Initial Launch Mode** window. If you don't see the type you want to use, you can click **Back** to return to this first window, select the appropriate launch mode, then click **Next** to see different types listed.

Table 1. Launch configuration types designed for QNX OS systems

Name	Description	Usable launch modes
C/C++ QNX Application	Supports launching a C/C++ application on a QNX target and then debugging, profiling, or analyzing that application using an IDE tool. This launch configuration type defines basic settings such as the project name, the application binary to run, and the build configuration to use. It also specifies properties for starting the application, including the command-line arguments and environment variables, as well as debugger settings.	Run, Debug, Memory, Check, Profile, Coverage, Attach
C/C++ QNX Attach to Remote Process	Allows you to attach an IDE tool (GDB, Application Profiler, etc...) to a process running on a target, so you can debug, profile, or analyze an application at runtime. The properties are similar to those for C/C++ QNX Application but don't include the controls for starting an application because in this case, you're attaching to a process that's already running.	Debug, Memory, Check, Profile, Coverage, Attach
C/C++ QNX Local Core Dump Debugging	Allows you to debug a core file stored on your host, whether that file came from a QNX target or another source such as a customer. The available properties include the basic settings, the paths of any shared libraries that were used by the crashed process and that you want to debug, and debugger settings. For this launch configuration type, the IDE uses a special launch target, **Local**.	Debug
C/C++ QNX Remote Core Dump Debugging	Allows you to debug a core file that was generated on the target when a process crashed. The IDE automatically creates a launch configuration of this type when you choose to debug the core file when prompted by the IDE after it detects the crash. The available properties include the basic settings, the paths of any shared libraries that you want to debug, and debugger settings.	Debug
Neutrino Kernel Event Trace	Controls how kernel event trace data is logged. You can define the trace duration, how the trace data is uploaded to the IDE, which kernel events are logged, and more.	Log

Name	Description	Usable launch modes
QNX File Transfer	Automates file transfers between the target and host machines. You can list files to be uploaded (copied from host to target) or downloaded (copied from target to host). For each transfer, the host file must be selected from either a workspace location or the local filesystem, and the target file must be specified as a relative path beginning with the target connection name. The Target File System Navigator lets you manually transfer files.	Run, Debug
Qt QNX Application	Supports launching Qt applications on a local or remote QNX target.	Run, Debug

Page updated: August 11, 2025

QNX perspectives

QNX Tool Suite | Integrated Development Environment User's Guide | Developer | Setup

Perspectives control which views are shown in the IDE and how they are laid out. The IDE contains many perspectives that support application and system analysis for QNX OS targets.

Here, we list only those perspectives and views designed for working with QNX targets. The default perspective is the C/C++ perspective, which is described in "Perspectives available to C/C++ developers" in the *C/C++ Development User Guide*. General information about perspectives is given in "Working with perspectives" in the *Workbench User Guide*.

To open a specific view, select **Window** > (and then)**Show View** > (and then)**Other**, then expand the entry for the current perspective and select the view that you want to open. You can also reset the perspective to its default views by selecting **Window** > (and then)**Perspective** > (and then)**Reset Perspective**.

Table 1. QNX perspectives and associated QNX views

Name	Description	Associated QNX views
QNX Analysis	Displays data produced by analysis sessions. The IDE opens this perspective when you launch an application in any of the **Coverage**, **Memory**, **Profile**, or **Check** launch modes. In multiple views, the IDE displays graphs and statistics based on analysis data. The individual tool perspectives used in earlier releases have been combined into this new perspective.	Analysis Sessions Execution Time Memory Backtrace Memory Events Memory Problems Valgrind
QNX System Information	Provides detailed realtime data about your target machine's resource allocation and usage, along with key information such as CPU usage, program layout, the interaction of different programs, and more. You should open this perspective when you want to examine the current state of your target.	APS View Connection Information Malloc Information Memory Information Process Information Signal Information System Resources System Summary Target File System Navigator Target Navigator
QNX System Profiler	Displays the results of kernel event traces and the statistics gathered during them. You can view visual data based on the kernel event log (**.kev**) files and gain insight into the system activity and process interaction that occurred during the trace period. The data read by the System Profiler is generated by the instrumented QNX OS kernel, **procnto*-instr**, then written to a log file by a data capture program on the target, before being read in and displayed by the IDE.	Client/Server CPU Statistics Condition Statistics Event Data Event Owner Statistics Filters General Statistics Target Navigator Thread Call Stack Thread State Snapshot Timeline State Colors Trace Event Log Why Running?

Page updated: August 11, 2025

QNX C/C++ Project properties

QNX Tool Suite | Integrated Development Environment User's Guide | Developer | Setup

For projects that use managed recursive makefiles, the build settings are defined through the UI and the IDE regenerates the makefiles whenever you change these settings.

NOTE:
In past releases, such projects were called simply *QNX Projects*. Because this term is vague and the preferred choice is to use standard (non-recursive) makefiles, these projects are now called *QNX Legacy Recursive Make projects*. This term is used in the QNX Project wizard.

When you're creating QNX Legacy Recursive Make projects, the QNX Project wizard exposes some of the build settings in its **Project Settings** dialog, allowing you to configure the new project. After you've created the project, you can access the **QNX C/C++ Project** properties to edit those same build settings and define new ones. For instance, you can add include paths to your project through the Compiler tab and libraries through the Linker tab; both tabs are shown only in the properties dialog (not the wizard). Also, you can enable code coverage through the **Options** tab.

You can access the QNX C/C++ Project properties in two ways:

- by left-clicking a project entry in the **Project Explorer**, choosing **Project** > (and then)**Properties** from the main menu area, then selecting **QNX C/C++ Project** on the left in the resulting window
- by right-clicking a project entry, choosing **Properties** in the context menu, then selecting **QNX C/C++ Project** on the left in the resulting window

Options

The **Options** tab contains these fields:

Category	Name	Description
General Options	**Share all project properties**	Whether to share all of your project's properties with other developers. By default, some properties (e.g., active targets) are local, meaning they're stored in the **.metadata** folder in your own workspace. If you check this box to enable sharing of all properties, the IDE then stores them in a **.cproject** file, which you can save in your version control system so that others may access that file.
Build Options	**Build for Profiling (Call Count Instrumentation)**	Makes the IDE insert code at the beginning of each function, to track how many times it's called and who calls it. After building an application with this option enabled, you can use the Application Profiler to collect statistics about call counts and callers.
Build Options	**Build for Profiling (Function Instrumentation)**	Makes the IDE insert code just after the entrance to and just before the exit from each function, to measure its total runtime. After building an application with this option enabled, you can use the Application Profiler to determine how much time is spent in each function and in its descendants (i.e., functions that it calls).
Build Options	**Build with Code Coverage**	Builds the project for code coverage, which makes the IDE insert code that counts the execution of basic blocks as well as generate notes (**.gcno**) files that map block count data to individual lines of code. When you then run an application with the Code Coverage tool, the IDE visually presents line-by-line coverage data.

Build Variants

This tab specifies the processor architectures and build variants to generate binaries for. In the list area, you can expand the architecture categories to see and enable specific variants.

NOTE:
Initially, none of the variants is enabled, but you must select at least one variant when creating the project. If you want any build variants to be auto-selected for new projects, you can change the default

preferences for the legacy QNX project type. To do so, open **Windows** > (and then)**Preferences** > (and then)**QNX** > (and then)**New Project** > (and then)**Build Variants**.

You can click the **Select All** button on the right of the list area to enable all of the listed variants, or the **Deselect All** button to disable all of them. After selecting an architecture in the list, you can click **Add** to add a new build variant; for example, you could create a build variant for unit testing. To remove a variant that you defined, select it in the list, then click **Delete**. Note that you can't remove the debug and release variants.

For each architecture, you can choose one build variant to be the "indexer", by clicking that variant and then the **Set Indexer Variant** button. This variant's symbols and include paths will then be used for source indexing. The impact on the C/C++ Editor is that the source indexing determines the macro definitions, inclusion/exclusion of additional code, the navigation to header files, and more.

Library

This tab is shown only for libraries, not standalone applications. In the QNX Project wizard, you must select the project type in the **Basic Settings**. If you choose any type other than **Application**, you'll see the **Library** tab in the QNX C/C++ Project properties after you've created the project.

The **Build target type** radio buttons let you select the kind of libraries to build. You can choose from these options:

Static library

Combines the object files into an archive (**lib*xx*.a**) that can be directly linked into executable programs. A static library is a collection of object files that you can add to applications.

Shared library

Combines binary objects and joins them so they're relocatable and can be shared by many processes. The shared object (**lib*xx*.so**) is an executable module that gets compiled and linked separately. Select this option if:

- you have code to reuse
- you want to generate a library that will be relocated (copied) to a target machine
- you want to statically link code into a shared object

When you create an application that uses a shared library, you must define your shared library's project as a Project Reference for your application.

If you choose to use versioning (in the General Options of the Linker tab), two files are generated for a shared library: **lib*xx*.so** and **lib*xx*.so.*n***, where *n* is a version number with a default of 1. The first file is a symbolic link to the latest version of the second file. If you don't use versioning, only one file (**lib*xx*.so**) is generated.

The IDE also generates a static shared library (**lib*xx*S.a**). This is a collection of objects compiled to be position-independent. The static shared library lets you build custom shared libraries that use a subset of the objects from the original library, so you can remove unused functionality and thus, reduce the overhead of the library.

Shared+Static library

Creates every kind of library that exports its symbols. This is the same as selecting **Static+Static shared library** (see the next option), except that the IDE also builds a shared object (**lib*xx*.so**).

Static+Static shared library

Generates two types of static libraries: A static library (**lib*xx*.a**), which is meant for linking into executable programs, and a static shared library (**lib*xx*S.a**), which has position-independent code (PIC) and is meant for linking into shared objects.

Shared library without export

Generates a shared library without versioning. This is useful for applications to discover functionality extensions at runtime (e.g., driver modules that plug into hardware). Generally, you write code to open the library with *dlopen()* and to locate specific functions with *dlsym()*.

General

This tab defines the following basic properties:

Name	Description
Installation directory	The directory where the `make` install process copies the binaries that it builds.

Name	Description
Target base name	The base name for the library or executable being built. This is the filename section between the **lib** prefix (for a library) and the extension delimiter (`.`). Typically, this is suffixed by patterns such as **_g** for debug, or **_foo** for a variant named `foo`, and so on. For more information about recursive makefile naming, see the Conventions for Recursive Makefiles and Directories chapter in the *Programmer's Guide*.
Use file name	The name of the `usemsg` file that puts the use message into the binary file. The message is kept in a header, and the `usemsg` command looks for this header to print the message.

Compiler

The **Compiler** tab contains these fields:

Category	Name	Description
General Options	**Compiler**	The compiler type. This field is read-only if the SDP installation that you're using has only one compiler installed. SDP 7 ships with only one compiler, with a type of GNU Compiler Collection (4.7.3). If you're using an SDP setup with multiple compilers, you can select from among multiple options. When you change the SDK selection to build programs with another SDP version, the options in the **Compiler** dropdown are updated based on that new selection.
General Options	**Warning Level**	Set the warning level threshold, which can be: • `Default` — use the compiler's default warning level • `Suppress` — don't output any warnings; equivalent to level 0 • `1` to `9` — output warnings at or above this level
General Options	**Generate Preprocessor Output**	Makes the preprocessor write out intermediate code for each source file. The output file is named ***source_file*.i** (for C) or ***source_file*.ii** (for C++).
General Options	**Optimization Level**	The optimization level of the binary, either **Default** (which is level 2 for `qcc` or a number from 0 (no optimization) to 3 (most optimization).
General Options	**Dependency checking**	Makes the preprocessor write out a dependency map for each header file. The output file is named ***source_file*.d**. You can check dependencies and generate maps for user headers only or for all headers.
General Options	**Debug Symbols**	Whether to generate debug symbols. This field is read-only in Regular mode and has **Default** selected, meaning that the inclusion of debug information in the outputted binary depends on the build variant. If you switch to Advanced mode, you can override this setting for some build variants, by selecting **With Debug Symbols**. This ensures that debug symbols are written into the binary, even if they normally aren't for that variant.
General Options	**Macro Definitions**	Provides a list of symbol definitions to pass to the compiler using the command-line option `-D` *name*[=*value*]. You don't need to type the `-D`; the IDE adds it automatically.
General Options	**Other Options**	Provides any other command-line options not already covered in the **Compiler** tab. For details about all compiler options, see the `q++, qcc` entry in the *Utilities Reference*.
General Options	**All Compiler Options**	Displays all command-line options (flags) that will be passed to the compiler based on the settings in other fields.

Category	Name	Description
Include Paths	**Include import directories**	A list of directories telling the compiler where to look for included files. You must add to the list any directory containing a user header file referred to by an `#include` statement. On the right, there are three buttons that open file selectors so you can add directory entries: • **Workspace** — Browses the workspace projects. The IDE uses relocatable notation, so even if other team members use different workspace locations, you can all work successfully without any additional project adjustments. • **QNX Target** — Starts browsing at the directory defined by the *QNX_TARGET* environment variable, allowing you to pick a directory from this area of the host filesystem. • **File System** — Lets you pick any directory on your host machine. Below this, the **Delete** button lets you delete entries and the **Up** and **Down** buttons let you change the order in which the include paths are searched. The order is important when multiple header files have the same name.
Extra Source Paths	**Source import directories**	A list of directories telling the compiler where to find source code (which is needed for displaying debug symbols) in locations other than the project root directory. The controls are the same as for **Include import directories** —you can add entries from the workspace, *QNX_TARGET* directory area, or local filesystem, you can delete entries, and you can adjust their order.

Linker

The **Linker** tab contains these fields:

Category	Name	Description
General Options	**Generate map file**	Whether to print a link map to the build console.
General Options	**Stack Size**	The stack size, in bytes or kilobytes. For kilobytes, you must put a K at the end. The number is specified in decimal.
General Options	**Export Symbols**	Defines the level of final stripping done by the linker for your binary. This can be the default setting or it can range from exporting all symbols, to removing only the debug symbols, to removing all symbols.
General Options	**Artifact Name**	The name of the binary file generated for your project. By default, an application executable has the same name as the project from which it's built; you must fill in this field to give the artifact another name. A library file has prefix of **lib** and an extension of **.a** or **.so**, depending on the type of library. The base name, which is in-between, contains the name given in this field (or the project name if the field is blank). Debug variants of application and library binaries have the **_g** suffix in the filename, just before the extension.
General Options	**Link against CPP library**	*(for QNX C++ projects only)* • **Default** — The standard QNX C++ library, with support for all standard C++ features (exceptions, STL, etc.) • **LLVM with exceptions** — The LLVM C++ library, with support for exceptions • **GNU with exceptions** — The GNU G++ Standard Library, with support for exceptions

Category	Name	Description
General Options	**Shared Library Name**	*(for build target types that generate a shared object (**.so**) library file)* Sets the internal name (SONAME) of the shared object. This setting doesn't affect the actual filename. Defining an SONAME allows applications to determine which library version they're dynamically linking to, as explained in "Specifying an internal name" in the *Programmer's Guide*.
General Options	**Shared Library Version**	*(for build target types that generate a shared object (**.so**) library file)* Sets the version number for both the internal name (SONAME) and the filename of the shared object. If the library doesn't have a version number (e.g., its filename is to be **lib*xx*.so** with no numeric suffix), select No. This way, the shared-object name isn't hard-coded in the library. To use versioning, you can leave the field setting as `Default`, in which case the IDE uses 1 as the version number, or select an exact version number within the available range. The filename is then **lib*xx*.so.*n***, where *n* is a number based on this setting. Also, the SONAME ends with **so.*n***. The loader requires the filename to be exactly like this because all dependent projects will refer to the library as **so.*n*** (in the `NEEDED` section of the executable).
General Options	**Other Options**	Provides any command-line options not already covered in the **Linker** tab. For details about linker options, see the `q++, qcc` entry in the *Utilities Reference*.
General Options	**All Linker Options (LDFLAGS)**	Displays all command-line options (flags) that will be passed to the linker based on the settings in other fields.
Libraries		The libraries to link to your project. Each library entry has three fields: • **Name** — The base name, which is the filename part without the **lib** prefix (which the linker adds automatically) or the extension (**.a** or **.so**) • **Type** — The library type, one of **Static**, **Dynamic**, **Stat+Dyn**, or **Dyn+Stat** • **Match build variant** — Whether the linker looks for a library version that matches the final binary's version. When `Yes`, the library name format must match that of the artifact being built based on whether the **_g** suffix is present. For instance, if you want to build a release binary version that uses a debug library version, you must set this field to `No`. On the right, the **Add** button creates a new entry, in which you must fill in the fields manually. The two buttons just below open file selectors so you can automatically fill in the fields by selecting a library file: • **Workspace** — Browses the workspace projects. The IDE uses relocatable notation, so even if other team members use different workspace locations, you can all work successfully without any additional project adjustments. • **QNX Target** — Starts browsing at the directory defined by *QNX_TARGET*, allowing you to pick a directory from this area of the host filesystem. Below this, the **Delete** button lets you delete entries and the **Up** and **Down** buttons let you change the order in which the libraries are processed. If a static library references symbols defined in another static library, the library containing the reference must be listed before the one containing the definition. If you have cross or circular references, you might not be able to satisfy this requirement.

Category	Name	Description
Libraries Path	**Library directory expression**	A list of directories telling the linker where to look for library files. This list must contain any directory storing an archive (**.a** file) or shared object (**.so** file) that implements library functions called by your project code. When you add an entry to the **Library** list, the library's path is automatically added to the **Libraries Path** list. However, deleting an entry from the first list doesn't delete it from the second one. You can explicitly add library paths by using the **Workspace** and **QNX Target** buttons, which behave the same as for the **Library** field. Meanwhile, the **File System** button lets you pick any host directory. The **Delete**, **Up**, and **Down** buttons also behave the same way.
Extra Object Files	**Extra objects or libraries**	The extra object or library files to link with your project. You can't manually fill in an entry but you can click the **Workspace**, **QNX Target**, and **File System** buttons to pick a file through a file selector. These buttons behave the same way as they do for **Libraries**. The remaining buttons are also the same—you can delete entries and adjust their order in the list. **NOTE:** The file selector may seem slow when adding files. This is because the IDE can't make assumptions about naming conventions, so it must inspect a file to determine if it's an object file or library.
Post-build Actions	**Commands**	Specifies commands to run after building the project. These commands are run sequentially, in the order listed in this field. If you click **Add**, the resulting dialog asks you to pick from four categories: **Copy result**, **Move result**, **Rename result**, and **Other command**. For the first three categories, extra fields let you specify what component you want to copy, move, or rename, and the destination or new name. For the last category, you must enter the command string in another field. You can remove a command with **Delete** and change the command order with **Up** and **Down**.

Make Builder

The **Make Builder** tab exposes these fields:

Category	Name	Description
Build Command	**Use default**	Whether the default `make` command is used for building your project.
Build Command	**Build command**	Defines a custom build command. It could be a `make` command with customized arguments or an invocation of another utility. In the latter case, you must provide the path of the build utility. This field is editable when **Use default** is unchecked. On the right, you can click **Variables** to open a window that displays `make` variables and allows you to pick one variable and provide an argument for it. The required string token is then added to the `make` command when you close this window.
Build Settings	**Stop on first build error**	Whether you want the IDE to stop building as soon as it encounters a `make` or compile error.
Workbench Build Behavior	**Build on resource save (Auto Build)**	After you check the box, you can change the name of the auto-build target, using the text field. Clicking **Variables** opens the same window used for **Build command**, so you can use variables in the target name.
Workbench Build Behavior	**Build (Incremental Build)**	The checkbox enables the text field so you can change the name of the target for incremental builds. The **Variables** button lets you add variables to the name.

Category	Name	Description
Workbench Build Behavior	**Clean**	The checkbox enables the text field so you can change the name of the target for cleaning. The **Variables** button lets you add variables to the name.
Build Location	**Build directory**	Defines the directory from which your project is built. By default, this field is blank, which means that the project root directory is used. You can enter a relative path in the workspace or click the **Workspace** button to select a path from the workspace. The **File System** button lets you pick any host directory, while **Variables** lets you add variables to the directory name.
Parallel Build Setting	**Use parallel jobs**	Enables parallelism in building, which reduces the build time but increases memory usage.
Parallel Build Setting	**Parallel job number**	The maximum number of parallel build jobs to allow. There's a limit on how much parallelism the IDE can achieve when building; this limit depends on many characteristics of your project, including interproject dependencies.

Error Parsers

In this tab, you can select the build output parsers to apply to the project and specify in what order to apply them. The output parsers scan the build results for error, warning, and information messages and generate problem markers, which visually indicate problems in views such as **Console** and **Problems**.

The **Error Parsers** tab lists all supported parsers and provides checkboxes for selecting or unselecting them, as well as the **Up** and **Down** buttons to change the application order for the parsers. There are also **Select All** and **Unselect All** buttons, for convenience.

Regular and Advanced modes

The **QNX C/C++ Project** properties dialog can appear in two different modes: regular and advanced. Regular mode lets you configure build settings at the project level; usually, this offers a sufficient level of control. But you can switch to advanced mode to override certain settings for particular build variants.

To switch modes, click the **Regular** or **Advanced** button at the bottom of the dialog (the button shown is for the mode currently *not* displayed). For new projects, the regular mode is displayed initially, but the IDE remembers your mode setting so it reuses the last mode whenever you reopen the dialog.

In advanced mode, the properties dialog displays an additional panel between the left-side navigation and the properties tabs. In this extra panel, the **Platform** and **Variant** dropdowns let you choose an architecture and build variant for which you can override some settings. When you select from both these dropdowns, the display is filtered to show only the Compiler and Linker tabs. The settings that you can override include but aren't limited to:

- Warning Level
- Optimization Level
- Debug Symbols
- Libraries or Library Paths

Page updated: August 11, 2025

Running an application

From the launch bar, you can run an application based on a launch mode and a launch target.

To run an application:

1. In the launch bar, click the button for the Launch Configuration dropdown, which is the middle dropdown.
2. Select the entry for the project that you want to run.
   The IDE generates a default launch configuration for each project. We use this default configuration in our example, but you can create your own configurations to support different scenarios. For more information, see "Managing launch configurations".
3. Click the button for the Launch Mode dropdown, which is the leftmost dropdown.
4. Select the launch mode based on your user goal (i.e., whether you want to run, debug, or profile the application).
   The launch mode determines the version of the application binary that gets run.
5. Click the button for the Launch Target dropdown, which is the rightmost dropdown.
6. Select the target on which you want to run the application.

   NOTE:
   Ensure that the connection to the launch target is active, by checking for a purple circle without a red square () next to the target name. If you see a red square, the target isn't connected and you must configure it for IP communication.
7. Click the Run button ().

The IDE first looks for an up-to-date version of the application binary. If it doesn't find one, the IDE builds the project and displays the output in the CDT Build Console pane of the **Console** window.

After a successful build, the IDE copies the binary to the target and starts running the binary. The program's output is sent to the **Console** window, in a separate pane from the build output. You should see "Hello World!!!" (or whatever text that you expect based on your code changes) in this new pane.

While the application is running, its process appears in the target machine's process list, which is seen in the **Target Navigator**.

You can terminate the application by pressing the Stop button () in the launch bar.

Page updated: August 11, 2025

Running memory-analyzing tools from the command line

QNX Tool Suite | Integrated Development Environment User's Guide | Developer | Setup

You can run the same tool used by Memory Analysis as well as any Valgrind tool from the command line, outside of QNX Momentics IDE. This is handy for analyzing applications during system startup or when there's no network connection to the target.

Running an application with **librcheck**

The Memory Analysis tool in the IDE displays the data produced by **librcheck** (the debug allocation library), which is implemented separately from the tool. From the command line, you can launch an application with **librcheck** loaded. For more information, see the "Using the **librcheck** library" section in the *Programmer's Guide*.

If you want to use the backtrace feature to see stack data, you must store **libunwind.so.8** in a target location accessible to *dlopen()*. For more information, see the *dlopen()* entry in the *C Library Reference*. The **librcheck** library still functions without **libunwind**, but prints a warning that it can't produce backtraces when tracing function calls.

To view the memory data generated by **librcheck**, you must either attach the Memory Analysis tool to one of the application's processes or wait until the application exits and then copy the trace file from the target to the host and import it into the IDE. Although the trace file is text-based, its contents are too cryptic to read manually; the IDE is the only practical way to view **librcheck** data.

Running a Valgrind tool on an application

You can issue a command that tells Valgrind to run Massif, Memcheck, or Helgrind on the specified application with certain options and environment variables. For examples of such command lines, see the `valgrind` entry in the *Utilities Reference*.

To see backtrace information (i.e., source file names and line numbers) in the analysis results, you must provide Valgrind with an application binary and libraries that contain debug symbols, or with access to the debug symbols in separate **.sym** files. In the latter case, the symbol files must be kept in the same directory as the libraries (and binary); using *LD_LIBRARY_PATH* won't work because Valgrind doesn't use the dynamic loader.

The symbol files for SDP components delivered as libraries (e.g., audio drivers, networking protocols, video capture) are available from the QNX Software Center. Some component packages contain the stripped libraries and the debug symbol files. For others, there's a separate package for the symbols. Typically, symbol packages are automatically installed (if you've kept the default QNX Software Center settings), but you should confirm that the symbols that you need have been installed. For more information, see the *QNX Software Center User's Guide*, available from the QNX Download Center (https://www.qnx.com/download/).

Valgrind produces a log file when it finishes executing the program. To see the analysis results, you must copy the log file from the target to the host, then import it into the IDE. Although the log file is somewhat readable, we recommend viewing the results in the IDE because it presents them in a more readable way and lets you navigate to the related source code.

Page updated: August 11, 2025

Measuring application performance from the command line

You can run an application built with profiling instrumentation or run Valgrind Cachegrind on an application from the command line, outside of QNX Momentics IDE. This option gives you more control over how an analysis tool is configured while still allowing you to view the analysis results in the IDE.

Running an instrumented application binary

After building an application binary with either call count or function runtime measurement instrumentation, you can copy the binary to the target. Then, you must set environment variables (as explained below) to direct the data output. At this point, you can run the instrumented binary to generate the profiling data.

Position sampling profiling

> **CAUTION:**
> Position sampling profiling doesn't work with an SDP 8.0 target. This section only applies if you're working with an SDP 7.0 or 7.1 target.

With this profiling method, you must set *PROFDIR* to the directory where the results are to be written. When you do so, the results file is stored at ***PROFDIR*/gmon.out.*pid*.*process_name***. If you don't set this environment variable, the file is named simply **gmon.out** (which makes it hard to distinguish from files produced by other profiling sessions) and is stored in the directory where the program ran.

> **NOTE:**
> The program must exit normally for profiling data to be written. If you don't see a results file, you must fix whatever problems are preventing the application from successfully terminating. The other option is to force it to exit normally by attaching a signal handler that calls *exit()*.

When used from the command line, this profiling method has the following restrictions:

- Information for individual threads isn't available — In the threads tree shown by the **Execution Time** view, the profiling results from all threads get combined into one entry.
- Results from shared libraries aren't available — The results shown by **Execution Time** contain runtime estimates for code in shared libraries but simply say `@unknown` under **Location**, so you don't know which functions the entries refer to.
- No sampling information is collected if you don't run as root — If you don't run the program as root, the process can't attach the necessary interrupt handler or thread to write out position sampling data. This means you'll see only call counts and not function runtime estimates in the results.

Function runtime measurement profiling

With this profiling method, you must set *QPROF_FILE* to the full path of the file for storing the results. If you define only a filename in this variable, the results file gets written to the directory where the program ran. If you don't define the variable, no results are output. Also, the filename portion must finish with a **.ptrace** extension so the file can later be imported into the IDE.

Profiling data get written even if the program doesn't exit normally. However, if this happens, some data may be lost because some buffers can't be flushed to the results file. You can force the application to exit normally by attaching a signal handler that calls *exit()*.

The restrictions of individual thread information and results from shared libraries not being available also apply to this profiling method (for details, see the explanation above for position sampling).

Application profiling with kernel event tracing

You can run an application binary built with function instrumentation while performing a kernel event trace, to capture the function entrance and exit events for that application. To do this, you must first set *QPROF_KERNEL_TRACE* to 1 (to make the kernel start logging events), then run `tracelogger` in the background (to make this utility start writing the logged events to a trace file). Note that *QPROF_FILE* must *not* be defined when the tracing variable is set to 1.

The `tracelogger` utility must be run for long enough that the application can be started and execute until its normal exit point, or long enough to generate valuable profiling data. Information on the command-line options for setting the output file and tracing time are given in the `tracelogger` entry in the *Utilities Reference*.

Viewing results

To view the profiling results, you must either attach the Application Profiler tool to one of the application's processes or wait until the application exits and then copy the results file from the target to the host and import it into the IDE.

Running Valgrind Cachegrind on an application

You can issue a command that tells Valgrind to run Cachegrind on the specified application with certain options and environment variables. For examples of such command lines, see the `valgrind` entry in the *Utilities Reference*.

To see backtrace information (i.e., source file names and line numbers) in the analysis results, you must provide Valgrind with an application binary and libraries that contain debug symbols, or with access to the debug symbols in separate **.sym** files. In the latter case, the symbol files must be kept in the same directory as the libraries (and binary); using *LD_LIBRARY_PATH* won't work because Valgrind doesn't use the dynamic loader.

The symbol files for SDP components delivered as libraries (e.g., audio drivers, networking protocols, video capture) are available from the QNX Software Center. Some component packages contain the stripped libraries and the debug symbol files. For others, there's a separate package for the symbols. Typically, symbol packages are automatically installed (if you've kept the default QNX Software Center settings), but you should confirm that the symbols that you need have been installed. For more information, see the *QNX Software Center User's Guide*, available from the QNX Download Center (https://www.qnx.com/download/).

Valgrind produces a log file when it finishes executing the program. To see the analysis results, you must copy the log file from the target to the host, then import it into the IDE. Although the log file is somewhat readable, we recommend viewing the results in the IDE because it presents them in a more readable way and lets you navigate to the related source code.

Page updated: August 11, 2025

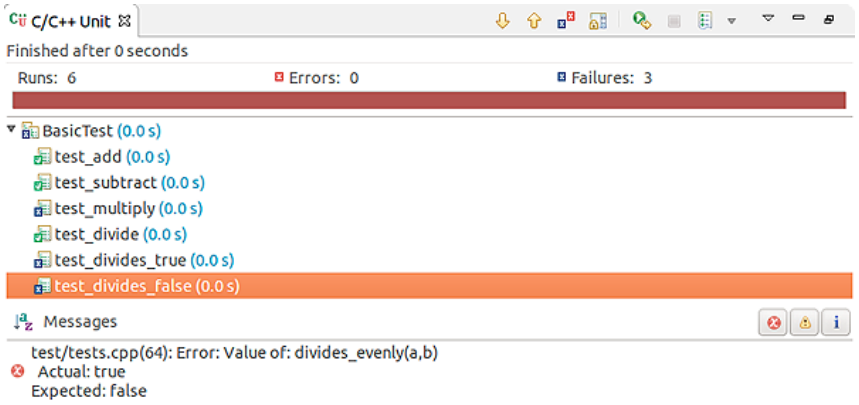
Running test programs

You can launch a unit test program on the target and the IDE will display the results of individual tests in a dedicated view as the program runs.

To run a test program:

1. In the Launch Configuration dropdown, select the project containing the relevant test program.
2. In the Launch Mode dropdown, select Run.
3. In the Launch Target dropdown, select the target for running the test program.
4. Click the Edit button (⚙) on the right of the Launch Configuration dropdown.
5. In the **Main** tab of the configuration editor window, enter the path of the test program in the **C/C++ Application** field.
6. In the **Arguments** tab, check the **Test Runner** checkbox to enable the test framework selector.
7. In the **Tests Runner** dropdown, select the framework on which your test program is based.
   The IDE can't auto-detect the framework you're using, so you must manually specify one.
8. In the **Upload** tab, check the **libgtest.so** and **libregex.so** checkboxes to upload these libraries to the target.
9. Click **OK** to save the configuration changes and close the window.
10. In the launch bar, click the Run button ( ).
    The IDE starts running the unit test program on the target. If necessary, the IDE builds the program before copying it to the target.

The **Console** view displays the raw output of the program while the **C/C++ Unit** view visually presents the test results (based on the IDE's parsing of those results):



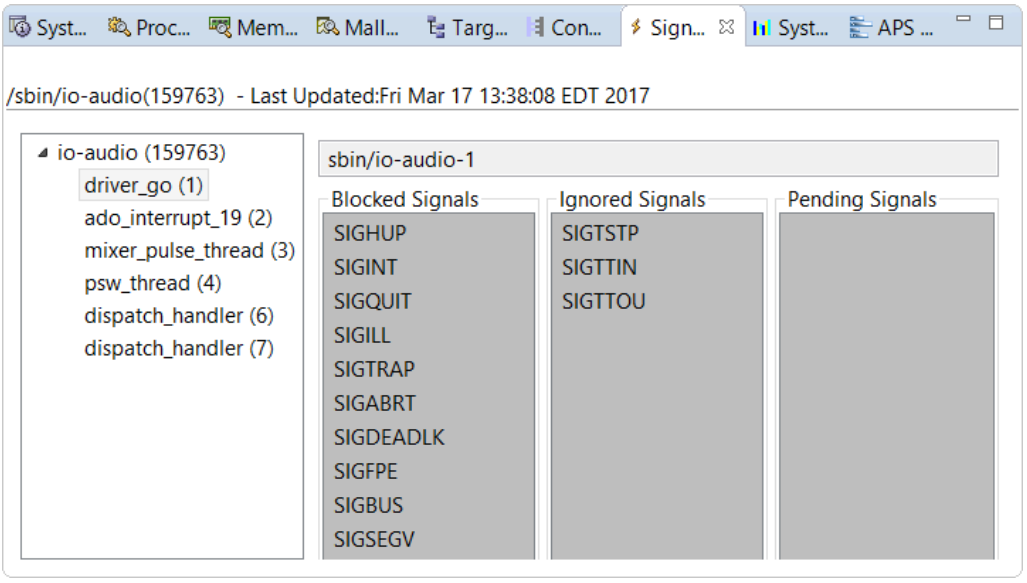
The IDE opens this latter view whenever it launches a test program; you can open the view manually by selecting **Window** > (and then)**Show View** > (and then)**Other** > (and then)**C/C++** > (and then)**C/C++ Unit**. At the top, the IDE illustrates testing progress (in a progress bar) and lists the numbers of completed tests (runs), errors, and test failures. In the area below, it lists the names and total running times of the program's test cases. The icon next to each test case name contains a green box with a checkmark if all tests in that test case passed or a blue box with an X if any test failed.

You can expand a test case entry to see the results of individual tests. When you click a specific test, the bottom area shows any messages output by the program while that test ran. There are also buttons on the right for filtering error, warning, and information messages.

Page updated: August 11, 2025

Signal Information

The **Signal Information** view lets you track the signals sent to the process selected in the **Target Navigator** view.



The signals are listed based on processing status:

- **Blocked Signals** — These signals are related to individual threads.
- **Ignored Signals** — These signals are related to the entire process.
- **Pending Signals** — These signals are related to the entire process.

You can click a thread entry in the left pane to see the signals blocked by that particular thread, ignored by the enclosing process, and pending for the process. Through the **Target Navigator**, you can send a signal to a target process. For example, you can terminate a process by sending it SIGTERM. In many cases, sending a signal terminates the process.

NOTE:
Unlike other System Information views, **Signal Information** doesn't automatically refresh. To refresh signal data for a process or thread, click the appropriate entry in the **Target Navigator**.

Page updated: August 11, 2025

Starting the IDE

QNX Tool Suite | Integrated Development Environment User's Guide | Developer | Setup

Depending on your development system OS, you can start the IDE from the system UI or the command line.

Unlike in previous releases, there's no shell script to launch the IDE. The environment is set by the installers, so you don't need to run a script that sets environment variables to point to various SDP installation directories.

To start the IDE:

- For Windows, choose **QNX Software Systems** > (and then)**QNX Momentics IDE 8.*X*** from the Start menu, where *X* contains the minor version number followed by the build number and timestamp, such as `0.00319T202003251340A`. You can also click the desktop icon with the same label, which is added when you install the QNX Tool Suite.
- For Linux, run ***IDE_base_dir*/qde**, where *IDE_base_dir* is where you installed the IDE package

> **NOTE:**
> This path is probably different from that for QNX SDP 7.0. In QNX SDP 7.0, the default path is **~/qnx/qnxmomentics/qde**, but in SDP 7.1 and 8.0, it's **~/qnxmomenticside/qde**.

Setting the workspace location

When you run the IDE for the first time, you can choose the workspace, which is the directory where the IDE stores your projects.

By default, this directory is **C:\Users*username*\ide-*version*-workspace** (on Windows), or ***$HOME*/ide-*version*-workspace** (on Linux). For Windows, *username* must not contain spaces.

When you start the IDE, you can tell it to store your workspace in another location, as follows:

1. When the **Eclipse Launcher** window appears, enter the new location.
   To do so, you can:
   - Manually enter a directory path in the **Workspace** text field. You can create a new workspace by entering a new path.
   - Click **Browse** and navigate to and choose a directory from the file selector.
   - Expand the **Recent Workspaces** dropdown and click a directory in this list.

   > **NOTE:**
   > The workspace path must not contain any spaces or non-standard characters. Although the IDE accepts them, the underlying build tools don't like directory and file names with such characters. For the list of unacceptable characters, see "Creating a QNX project".

   When creating a new workspace, you can copy the Workbench Layout and/or Working Sets settings from the current workspace, by expanding the **Copy Settings** dropdown and checking the corresponding boxes.
2. If you always want to access the same workspace location on startup, check the box labeled **Use this as the default and do not ask again**.
3. Click **Launch** to continue loading the IDE.

You can switch workspaces at any time by selecting **File** > (and then)**Switch Workspace**. The resulting submenu lists recently used workspaces and the **Other...** option, which opens the **Eclipse Launcher** window.

The workspace directory should never be checked into a revision control system nor located on a shared drive (unless you're certain that there will only ever be one person using it). The frequent and large-scale development cycles in workspace metadata might cause poor performance on network filesystems, particularly with large workspaces.

Where workspace files are stored

In the workspace directory, the IDE stores personal usage and project information in the **.metadata** subdirectory. The **.log** file in this subdirectory contains an internal error log, which is used to troubleshoot the IDE.

The **.metadata/.plugins** subdirectory stores your perferences so the IDE configuration is saved between usage sessions. This location also stores sessions generated by the analysis tools used by the IDE.

Page updated: August 11, 2025

Switching between SDKs

QNX Tool Suite | Integrated Development Environment User's Guide | Developer | Setup

QNX SDP provides a Software Development Kit (SDK) for writing applications for QNX OS. In the QNX Momentics IDE, you can change the SDK selection to use a toolchain from a particular QNX SDP installation.

In addition to the SDK from QNX SDP 8.0, this release of the IDE can work with the SDK from QNX SDP 7.0 and 7.1. Thus, you can install and use more than one of these platform versions on the same host, and you can import workspaces and projects created with the SDK from any of these versions.

To import workspaces, you must use the mechanism to switch workspaces and select one created by an earlier SDK version. The IDE will warn you that it will update the workspace, which may make it incompatible (and hence, non-usable) with the earlier version. If you continue and the workspace import succeeds, all projects in the existing workspace should appear in the **Project Explorer**. For instructions on importing individual projects, see "Importing existing projects" in the *Workbench User Guide*. The imported projects preserve all custom settings and are buildable in the new IDE.

NOTE:
When you run applications built with a particular SDK version on the host, the target must be based on the exact same version. Otherwise, many features of debugging (e.g., stepping through code at startup) don't work properly.

Global QNX Preferences

The **Global QNX Preferences** window is accessed through **Window** > (and then)**Preferences** > (and then)**QNX** and lets you specify which SDK to use when developing an application.

The **SDK Selection** fields set some environment variables according to the SDK that you select. The IDE uses these environment variables to locate files on the host computer.

The **Select SDK** field lists the available SDK installations. When you run the IDE for the first time, by default, it uses the last installed SDK that appears in this list. After you select an SDK, the IDE remembers the setting. At all times, the other fields have values based on this first field:

Field	Description
Select SDK	The name of the SDK that you want to use, or **Use Environment Variables** if you want to use the one specified by *QNX_HOST* and *QNX_TARGET*.
Version	The version of the QNX Tool Suite.
SDK Tools Path	The location of host-specific files.
SDK Platform Path	The location of target-specific files on the host machine.

The Select SDK field corresponds to QNX SDP installations. You can have multiple SDP installations on a system with different packages installed in each; make sure that you select the one that has the appropriate workspace, context, and targets for your project.

For example:

- To build a QNX VMWare target, the SDK selection must correspond to the SDP installation that has the PCI HW Module x86 package installed.
- To build a QNX VMWare target with graphics, the SDP installation you select must include the Screen Board Support VMWare (vmwgfx) package.

Environment variables used to locate files

When you select **Use Environment Variables** in the **Select SDK** field, QNX OS uses the following environment variables to locate files on the host computer:

QNX_HOST
: The location of host-specific files.

QNX_TARGET
: The location of target backends on the host machine.

QNX_CONFIGURATION

(QNX SDP 6.6 or earlier)

The location of the **qconfig** configuration files. These files are copied to the host by the QNX SDP installer, and indicate where you've installed the product.

QNX_CONFIGURATION_EXCLUSIVE

(QNX SDP 7.0 or later)

The location of the **qconfig** configuration files. These files are copied to the host by the QNX SDP installer, and indicate where you've installed the product.

You need to set this variable only if the default location isn't accessible or if the configuration files are located on a build server.

MAKEFLAGS

The location of included ***.mk** files.

TMPDIR

A directory used for temporary files. The `gcc` compiler uses temporary files produced in one compilation stage as the input to the next stage; for example, the output of the preprocessor is the input to the compiler.

Page updated: August 11, 2025

Building QNX OS Images

QNX Tool Suite | Integrated Development Environment User's Guide | Developer | Setup

Using the IDE's QNX System Builder feature, you can generate *OS images*, which are bootable images that contain startup code, the QNX OS, your applications, and any data files used by your applications. The IDE supports image building by allowing you to transfer an image to a target board.

Here are the main tasks for using the IDE to set up an OS image on a target board:

1. Create a QNX System Builder project to generate a system image for your target. This process is simple if a Board Support Package (BSP) exists for the board. If an exact match isn't available, you may be able to modify an existing BSP to meet your needs.
2. Modify your project as needed using the buildfile editor. You may skip this step during the first iteration just to build a basic image based on a BSP.
3. Build your project to generate the image files.
4. Transfer the image to your target board using a serial link or another method.

After you've booted the image and verified that the OS runs successfully on your target hardware, you might go through more iterations of this process as you optimize your system. For example, you could modify the project again to add components. You would then need to repeat the last three steps to run the new system image.

The IDE also lets you import BSP archives as projects and build them into system images. BSP archives contain the hardware-specific components needed to support the QNX OS on a particular board. The *Building Embedded Systems* guide provides an overview on BSPs. After you successfully build a BSP project, the output image files contain everything needed to boot the system and run the OS and any included applications.

Page updated: August 11, 2025

BSP filename conventions

In our BSP documentation, buildfiles, and scripts, we use a filename convention that relies on prefixes and suffixes to distinguish between file types. The names for Image Filesystem (IFS) files take the form **ifs-*board_name*.elf**, and the names for Embedded Filesystem (EFS) files take the form **efs-*board_name*.raw**.

The IDE uses a somewhat simplified filename convention. Only a file's three-letter extension, not its prefix or any other part of the name, determines how the IDE handles the file. For example, an OS image file is always an **.ifs** file in the IDE, regardless of its format: ELF, binary, SREC, etc. To determine a file's format, you need to view the file in an editor.

NOTE:
For a complete list of BSP filename conventions, see the "BSP structure and contents" section in *Building Embedded Systems*.

Page updated: August 11, 2025

Building an OS image

To build an OS image, you need to build the QNX System Builder project that contains the relevant buildfile. But unlike for other projects, the IDE doesn't create launch configurations for QNX System Builder projects, so you can't build them through the launch bar. Instead, you can use menu actions.

You build OS images simply by selecting the relevant project in the **Project Explorer** and choosing **Project** > (and then)**Build Project**. Or, you can right-click the project entry and choose **Build Project**.

The **Console** view shows the output from the build attempt. Often, it's helpful to see only the messages related to the last build operation. Before building, you can clear the view by clicking the **Clear Console** button () in the view toolbar.

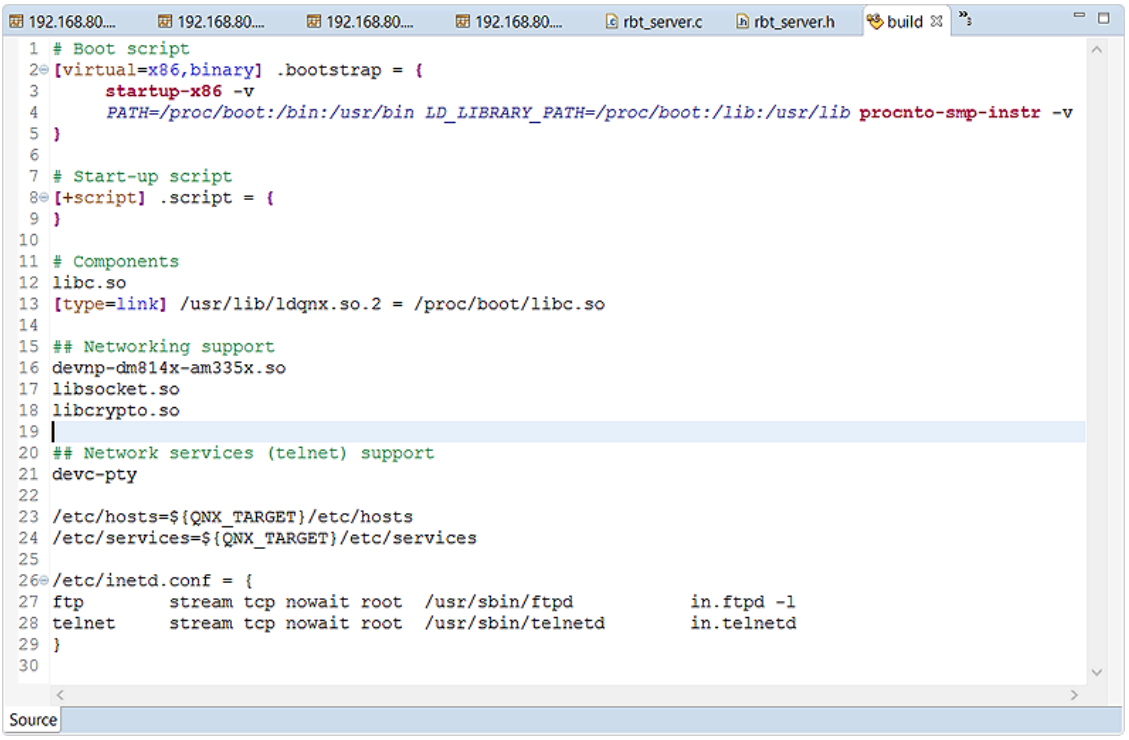
Page updated: August 11, 2025

Modifying an OS image

Using the buildfile editor, you can manually add any required components to your OS image.

NOTE:
If you have a BSP for your target board, you may first want to try booting the original, unmodified image on your hardware, to see if the image runs well. In this case, you can skip the modification step and just build the image. Later, you'll need to edit the buildfile when you're ready to add or change components in the image.

When you add a component, the IDE doesn't automatically add the shared libraries required for runtime loading. For example, if you add a network application to a QNX System Builder project, the IDE doesn't include **libsocket.so**; you must add it manually to ensure that Telnet can run. Also, the IDE doesn't automatically include the necessary DLLs, so you must also do that yourself.



NOTE:
To learn about buildfile syntax, see the OS Image Buildfiles chapter in *Building Embedded Systems*.

Page updated: August 11, 2025

Overview of OS images

QNX Tool Suite | Integrated Development Environment User's Guide | Developer | Setup

Before you use the IDE to create OS images for your hardware, you must first understand the operations that occur when a system starts up. You must also become familiar with the concepts involved in building images.

We recommend that you read the following sections in the *Building Embedded Systems* guide:

To learn about:	**See:**
The QNX startup sequence	"The boot process"
Image components for each stage of startup, including IPL, IFS, and EFS	"BSP components"
Startup programs	"Startup Programs"
OS images	"OS Images"
Combined images that join together multiple components	"Combining multiple image files"
Buildfiles, including buildfile syntax, bootstrap files, and startup scripts	"OS Image Buildfiles"

Page updated: August 11, 2025

Downloading an image to your target

The easiest way to transfer an image to your target is using a *ROM monitor*, a simple program that runs when you first power on the board. The ROM monitor lets you communicate with your board via a command-line interface (over a serial or IP link), so you can download images to the board's system memory and burn images into flash.

> **NOTE:**
> If your board doesn't have a ROM monitor, you probably can't use the download services in the IDE, so you have to get the image onto the board in some other way.

The IDE provides two mechanisms for transferring image files: serial terminals or the TFTP server.

Opening a serial terminal

The IDE includes a **Terminal** view so you can talk to your target without having to leave the IDE and use an external program like HyperTerminal.

To open a serial terminal and talk to your target:

1. Connect your target and host with a serial cable.
2. Select **Window** > (and then)**Show View** > (and then)**Other...**.
3. Expand **Terminal**, then select **Terminal** and click **OK**.

   The IDE displays the **Terminal** view, typically in the bottom right area.
4. Click the Open Terminal button ( ) in the view toolbar to open the **Launch Terminal** window.
5. In the **Choose terminal** dropdown, select `Serial Terminal`, then select your communications settings (e.g., serial port, baud rate) and click **OK**.

You can now interact with your target by typing in the view and transfer files with the **Send File** button, as explained in the next section.

Transferring files over a serial connection

In the **Terminal** view, the **Send file** button ( ) becomes active as soon as you open a serial connection with the target.

To transfer a file:

1. Using either the **Terminal** view or another method (outside the IDE), configure your target so that it's ready to receive an image. For details, see your hardware documentation.
2. In the view toolbar, click the **Send file** button ( ).
3. In the Data Transmission dialog box, below the File to transfer box, select the file by doing one of the following actions:
   - Click **Workspace** and in the File Selection dialog box, select a file and click **OK**.
   - Click **Filesystem** and in the File Selection dialog box, select a file from your host filesystem and click **Open**.
4. In the Select transfer protocol area, select **Transfer a file using the QNX sendnto protocol** or **Transfer raw binary data over the connection**.

   > **NOTE:**
   > The QNX `sendnto` protocol sends a sequence of records, including the start record, data records, and a go record. Each record has a sequence number and checksum. Your target must be running an IPL (or other software) that understands this protocol.
5. Click **Finish**.

The IDE begins transferring the file over the serial connection. To stop the transfer, you can click the stop button ( ) next to the progress bar.

Transferring files over TFTP

The IDE has a TFTP server for transferring image files to the target. This feature eliminates the need to use an external service for downloading images (assuming your target supports TFTP downloads). The TFTP server searches all QNX System Builder projects for system images whenever it receives download requests from a bootloader program.

To use this feature, your target must be configured for IP communications. Before powering on your target board (and hence, before its bootloader program runs), you must start the TFTP server. You can configure the IDE to automatically start it, by selecting **Window** > (and then)**Preferences** > (and then)**QNX** > (and then)**TFTP Server** and checking **Start TFTP Server when IDE opens**.

> **NOTE:**
> If you're running as a non-root user on Linux, you must run the server on a port higher than 2015. To set the port, in the same **TFTP Server** dialog, enter a number in the **Port** field.

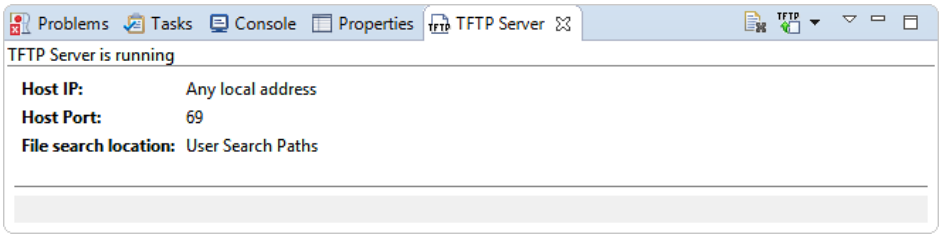
When you're finished configuring the TFTP server, click **OK** to save the settings and close the dialog.

If necessary, you can manually start the server:

1. Select **Window** > (and then)**Show View** > (and then)**Other** > (and then)**QNX System Builder** > (and then)**TFTP Server**.
2. Click **OK** to open the view. By default, it appears at the bottom right.
3. In the view toolbar, click the View Menu dropdown (▽), then click **Start**.

   For the remainder of the IDE session, the TFTP server listens for incoming transfer requests and fulfills them.

The internal TFTP server handles the requests, while the view shows the host IP address and port on which the server listens for them and the location where it looks for image files.



The view also provides the status of current and past transfers, by displaying the requested filename, a progress bar, and a status message during each transfer (in the grey area below the host information). You'll see data in this area just after you power on your board and its bootloader begins downloading image files. You can clear the view of all completed transactions by clicking the **Clear** button ().

The TFTP server recognizes files in the **images** directory of all open QNX System Builder projects, so you don't need to specify this path. However, the IDE deletes the contents of this directory during builds, so if you want to transfer files that aren't generated by the IDE, you must configure a new path:

1. Select **Window** > (and then)**Preferences** > (and then)**QNX** > (and then)**TFTP Server** > (and then)**User Search Paths**.
2. Click **New** to open the **Add New Search Path** window, then manually enter a directory or click a button to select a workspace or filesystem location from a file selector.
3. Click **OK** to save the settings and close the preferences window.

   The TFTP server is now aware of the contents of your selected directory.

Other image transfer methods

If your target board doesn't have an integrated ROM monitor, you may not be able to transfer your image over a serial or TFTP connection. In this case, you must use another method, such as:

- SD Card — copy the image to an SD Card plugged into your host, then plug the card into your target
- Flash programmer — manually program your flash with an external programmer
- JTAG/ICE/emulator — use such a device to program and communicate with your target

To learn how to connect to your particular target board, consult your hardware and BSP documentation.

Page updated: August 11, 2025

Types of images you can create

You can create bootable image filesystems, flash filesystems, or a combination of the two.

The IDE helps you create the following image types:

OS image (.ifs file)
: An image filesystem. A bootable image filesystem holds the `procnto` module, your boot script, and possibly other components such as drivers and shared objects.

Flash image (.efs file)
: A flash filesystem. (The “e” stands for embedded.) You can use your flash memory like a hard disk to store programs and data.

Combined image
: An image created by joining together multiple components (IPL, OS image, embedded filesystem image) into a single image. You might want to combine an IPL with an OS image, for example, and then download that single image to the board's memory via a ROM monitor, which you could use to burn the image into flash. The filename extension of a combined image indicates the file's format: **.elf**, **.srec**, etc.

If you plan to debug applications on the target, you must include `pdebug` in **/usr/bin**. If the target has no other forms of storage, include `pdebug` in the OS image or flash image.

Page updated: August 11, 2025

Creating a QNX System Builder project

To build an OS image, you need to define a QNX System Builder project. The IDE provides a wizard to set up this project based on your buildfile preferences.

To create a new QNX System Builder project:

1. Select **File** > (and then)**New** > (and then)**Project...**, or simply click the New button () to the right of the launch bar.
2. Expand **QNX System Builder**, then select **QNX System Builder Project** and click **Next**.
3. Type a name in the **Project name** field.

   NOTE:
   The project name must not contain any spaces or non-standard characters. The `make` utility (which is used for building) doesn't like directory and file names with such characters. Although the IDE might accept them, the build won't work.

   In the same dialog, you can also override the project storage location and specify the working sets that you want the project to belong to. When you're finished filling in the fields, click **Next**.
4. In the **Build File Initialization** dialog, choose from the following options:

   Create a new buildfile
   To create a simple buildfile, select your target platform from the dropdown list.

   Import from a BSP project
   To use an existing BSP project buildfile, select the buildfile from the dropdown list.

   Copy an existing buildfile
   Click **Browse...** to locate an existing buildfile. Refer to your BSP User's Guide to learn the name and path of the **.build** file for your board.

   NOTE:
   Creating a buildfile requires a working knowledge of boot script grammar, as described in the `mkifs` utility reference and the *Building Embedded Systems* guide.
5. Click **Next** if you want to customize the buildfile; the following section discusses customization options. Otherwise, click **Finish** to create the project. In the latter case, the IDE displays a new listing for the project and opens the buildfile for editing.

Creating a generic buildfile

If you chose to customize the buildfile, you can use the wizard's **Build File Template** dialog to create a generic minimal buildfile:

1. In the **Boot file** dropdown in the **Generic build file** panel (which is at the bottom of the window), select the format of the bootstrap file.

   This file calls the startup script, which sets up the environment and transfers control to `procnto`, which is the QNX OS microkernel and process manager. To learn how image components work together during startup, see "The boot process" in *Building Embedded Systems*.

   The format affects the `virtual` attribute of the `.bootstrap` entry found near the top of the buildfile.
2. In the **Startup** dropdown, select the type of startup script.

   This setting determines the name of the startup script and thus, the `.bootstrap` file contents.
3. Click **Next** if you want to create a combined image; this step is explained in the next section. Otherwise, click **Finish** to create the project and open its buildfile for editing.

Creating a combined image

You can create *combined images* by joining together several components—an Initial Program Loader (IPL), an OS image, and one or more flash images—into a single system image. For example, you can combine an IPL with an OS image and download the resulting image to the target board through Uboot.

The wizard's **Image Combination Details** dialog lets you control how images are combined with your QNX System Builder project. Under the **Build Image** panel, you can fill in the following fields:

Image name

The image name. By default, the IDE uses the same name assigned to the project.

Staging

The location where the IDE looks for files that go into the image. You can enter a filesystem location or click **Browse...** to browse available BSPs for a staging directory.

If you check the **Combine image with IPL** box, the IDE enables the following fields:

IPL file

The fully qualified name of an IPL file that you want to prepend to (i.e., concatenate at the front of) an IFS. You can enter a filename or browse available BSPs for an IPL file.

Pad IPL to

A dropdown list that lets you choose the amount of padding required for the IPL file that you want to prepend. Select a value equal to or greater than the size of the IPL. The IPL and the padding provide the start of the IFS.

> **CAUTION:**
> If the padding is less than the IPL size, the image won't contain the complete IPL.

If you want to burn the image into ROM at a given memory location, check **Create ROM image**. The IDE enables the following fields:

Offset

A board-specific offset, in hexadecimal, that indicates the distance from the beginning of the image to where the IFS starts. You typically use this setting for S-Record images.

Size

A numeric value for the ROM size. To determine the amount of ROM you need, compile the code to create a HEX file version of it; this file's size is the amount of ROM needed.

Format

A dropdown list of image formats. If you want to download an image to the target, the resulting ROM image will be copied to the format you choose.

When you're finished filling in these fields, click **Next** if you want to add project references; this step is explained in the next section. Otherwise, click **Finish**.

Adding project references

The wizard's **Project References** dialog lets you add project references to your QNX System Builder project. Simply check the boxes for any existing projects on which your new project depends, then click **Finish**.

What happens when you click Finish

When you click **Finish** in the QNX System Builder Project wizard, the IDE:

- Creates the project (including the necessary buildfile and makefile) and makes it visible in the **Project Explorer**.
- Opens the buildfile in the editor pane.

Page updated: August 11, 2025

Project layout

A QNX System Builder project can contain your **.ifs** file and multiple **.efs** files, as well as your startup code and boot script. You can import the IPL from another location or you can store it inside the project directory.

By default, a QNX System Builder project includes the following parts:

- **images/** directory — contains images and files created by the IDE when you build your project, and a **Makefile**
- **src/** directory — contains the resulting buildfile
- **.project** file — stores information about the project, such as its name and type; all IDE projects have a **.project** file

Page updated: August 11, 2025

System Information

The QNX System Information perspective displays realtime statistics about a target machine and the processes running on it. This perspective contains many views that provide diverse and useful information about memory, CPU, and resource usage, and about process and thread interaction on the target.

You can switch to this perspective at any time to examine the current state of your target and its applications. The statistics come from the general-purpose process-level allocator, so there's minimal overhead and you don't have to build an application with instrumentation code to see its runtime metrics. This design makes System Information the best starting point for detecting memory and performance issues and for understanding interprocess and interthread communication.

The perspective also lets you transfer files between the host and target machines and interact with targets and processes in other ways.

Unlike other integrated tools, System Information doesn't require any user action to enable and use. Also, it doesn't have an export or import feature for storing and sharing results outside of the IDE. The idea is that you monitor activity on the target and based on what you see, pick a more specialized tool to further investigate some aspect of the system or an application's behavior.

How data are displayed

Some System Information views—System Summary, System Resources, and APS View—provide system-wide statistics for the target selected in the **Target Navigator** view. Other views show process-level statistics, also based on the **Target Navigator** selection. These views are:

- Connection Information
- Malloc Information
- Memory Information
- Process Information
- Signal Information

When you click a target or process in the **Target Navigator**, the other views are refreshed. Multi-select is supported, by holding the **Ctrl** and/or **Shift** keys when selecting processes, similar to the functionality of Windows Explorer. However, only one other view, **Process Information**, can show data for multiple processes; the other views show data only for the selected target or process that's listed first (highest).

Reducing the overhead of data refreshing

When new process data become available, the content of the System Information views is automatically updated. This constant updating activity can somewhat increase the load on the host CPU, slowing the IDE. You can improve performance by:

- closing the QNX System Information perspective when you're not using it
- closing unneeded views within the perspective (you can reopen all of the default views by selecting **Window** > (and then)**Reset Perspective**)
- minimizing or hiding unneeded views
- disabling auto-refresh or reducing its frequency, through the **Target Navigator** toolbar (by default, the views are refreshed every five seconds, but you can reduce this rate)

Page updated: August 11, 2025

Client/Server CPU Statistics

This view lets you trace server time back to individual clients, by showing, for each thread, the time that it spent performing work for itself (*self time*) or that another thread spent performing work for it (*imposed time*).

In a message-passing system such as QNX OS, sometimes a particular thread consumes a lot of CPU time but that time is primarily spent servicing client requests. In this case, the total CPU usage for that server thread doesn't tell the whole story because you need to know which clients are imposing the heaviest load so you can optimize them. To identify clients and servers, the System Profiler uses the message-passing events in the kernel event trace log. For an explanation of self time versus imposed time in relation to such events, see "Isolating client and server CPU loads".

Every statistics view in the System Profiler perspective has these common behaviors:

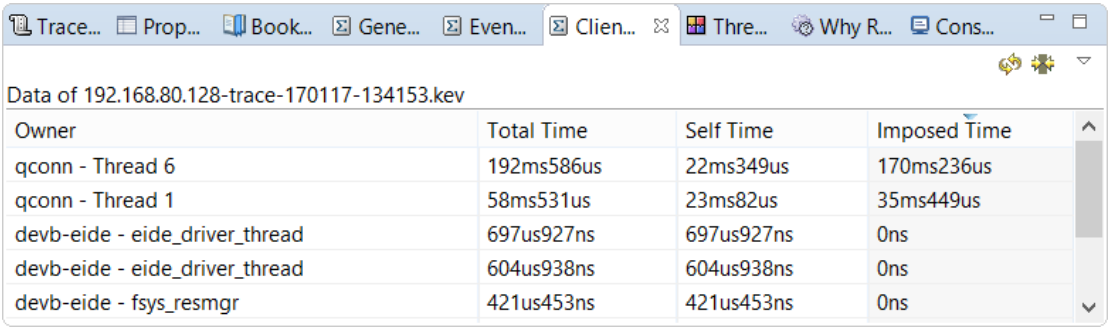
- No trace data are displayed until you click the **Refresh statistics** button (); some views have a different name for this button but it does the same thing.

  > **NOTE:**
  > If you're viewing statistics from one kernel event log file but then open another log file, the results in the statistics-related views won't reflect the newly opened file until you refresh them manually.

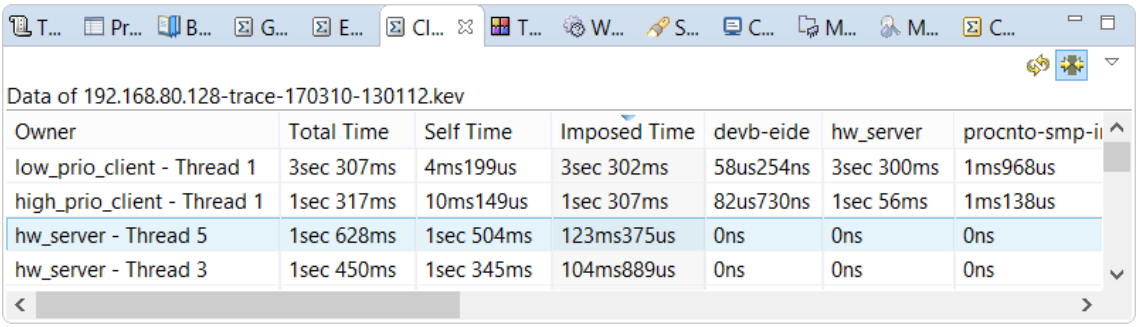
- They can display statistics that reflect either the entire trace period or the timeframe selected in an editor pane. The display mode is controlled by the **Toggle global/selection statistics gathering** button () as well as the top option in the upper right dropdown ().
- They can generate a CSV report of the data currently displayed. This feature is accessed through the bottom option in the upper right dropdown.

The **Client/Server CPU Statistics** view uses a table to display various times that threads spent in the Running state (this is slightly different from pure CPU usage). By default, four columns are shown:

- Owner — process name, followed by thread name
- Total Time — total of self and imposed times
- Self Time — overall CPU time the thread spent executing code for itself
- Imposed Time — overall CPU time that other threads spent executing code on behalf of the thread (in response to requests it issued by message sends); this accounts for all server threads



You can expand the table to show how much CPU time each client imposed on individual servers. To do so, click **Show all times** in the upper right dropdown. The table then displays columns containing server names and listing the times imposed by each client. This makes the table quite wide, so you'll likely have to scroll right to find the metrics of interest.



Note that imposed time is cumulative: if client A sends to server B, then until B replies to A, any CPU time that B consumes is seen as imposed time for A. If during that time B sends to server C, then server C's CPU time is also billed as imposed time for A. The rationale here is that B would not have engaged with C if A hadn't sent the initial message to B.

To sort the table by a metric, click its column header once for descending order or twice for ascending order. Sorting the threads by `Imposed Time` lets you quickly identify which clients are predominantly driving the system and which servers may be bottlenecks.

NOTE:
To see separate self and imposed CPU times, the trace must include all state change and message events (e.g., Ready, Running, Send Message, Receive Message, Reply, and Error). If you don't see these metrics, ensure that those events are enabled under the **Event Filters** tab in the log configuration. State change events are found under the **Process and Thread** class while message events are found under **Communication**.

Page updated: August 11, 2025

Condition Statistics

This view lets you see the statistical distribution over time of events matching certain conditions. The results include a table displaying the precise number of matching events at exact times during the trace as well as a bar graph illustrating how the number of events changed over time.

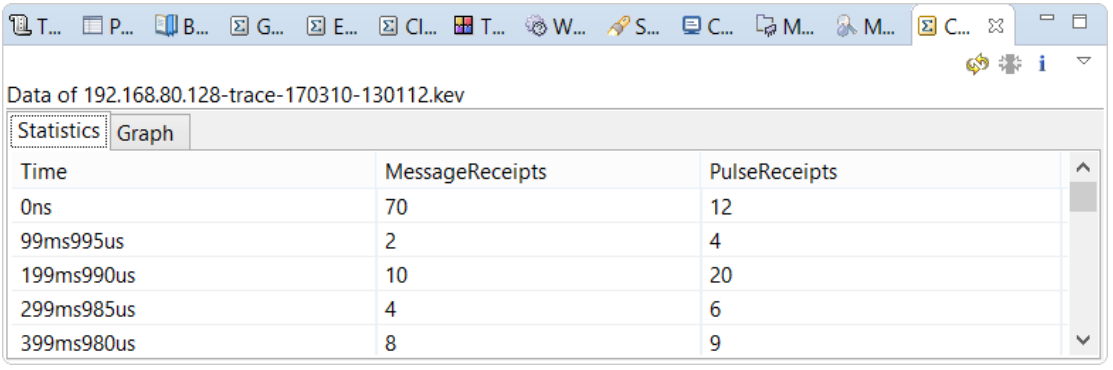
The conditions that you can use to gather event statistics are the same as those defined for the Timeline search feature. The **Condition Statistics** view lets you select any combination of these conditions.

Unlike with other statistics views, to see any data in **Condition Statistics**, you must first select some conditions instead of simply pressing the **Gather statistics** button ( ). When you first open the view, it displays a message advising you to pick the conditions, as well as two buttons:

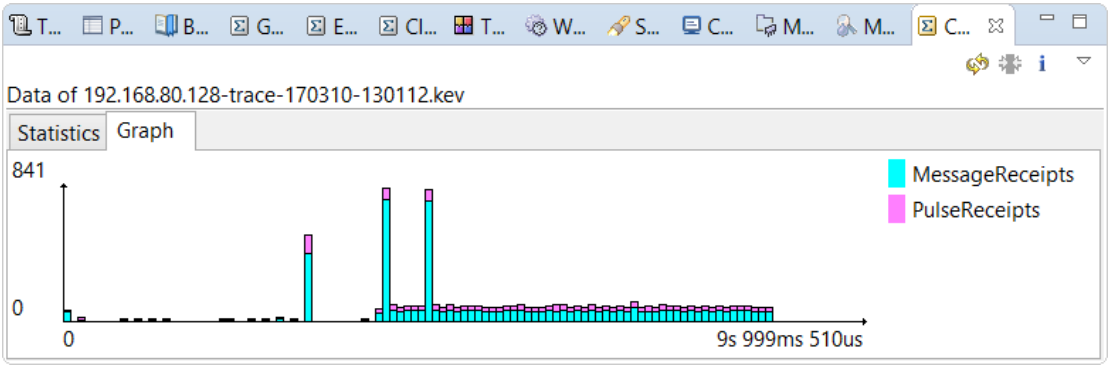
- **Configure Table** — opens a window for picking the conditions that events must match to be counted in the table statistics
- **Configure Conditions** — opens a window that lists all defined conditions and lets you add, remove, and edit them; this is similar to the bottom part of the **Trace Search** tab.

You can click the second button and modify the conditions to suit your event-matching criteria. To generate statistics, click the first button, check the boxes (in the **Condition Selection** window) for all conditions that you want the event data to be based on, then click **OK**. The IDE then prompts you to rerun the statistics gathering. In fact, you can manually update the table data any time after you've selected new matching conditions, by clicking **Gather statistics**. For convenience, the view toolbar has a **Configure Table** button ( ).

The results are shown in two tabs. The left tab, **Statistics**, is selected by default and lists how many events of each type occurred during specific trace intervals. These intervals are indicated by their start times given in the leftmost column. Note that you can resort the table by clicking column headers.



The right tab, **Graph**, displays a graph in which each bar indicates the number of events that occurred during a time interval. Each bar is sectioned into colored areas that represent different event types; the color codings are shown on the right.



The view offers the two common statistics features of toggling between global and selection-based statistics and generating CSV reports; both features are accessible in the upper right dropdown ( ). The dropdown also has the **Configure Table** and **Configure Conditions** options, as well as **Set number of divisions**, which lets you specify the number of statistical samples. This setting therefore determines how many rows are seen in the **Statistics** tab and how many bars are drawn in the **Graph** tab.

The higher the number of divisions, the smaller the intervals used for generating statistics. For instance, if the trace lasted exactly ten seconds and you use the default of 100 divisions, the samples reflect the number of matching events in 10 / 100 = 0.1 second-intervals. But you could increase this to 1000 divisions if you wanted to

use 0.01 second-intervals.

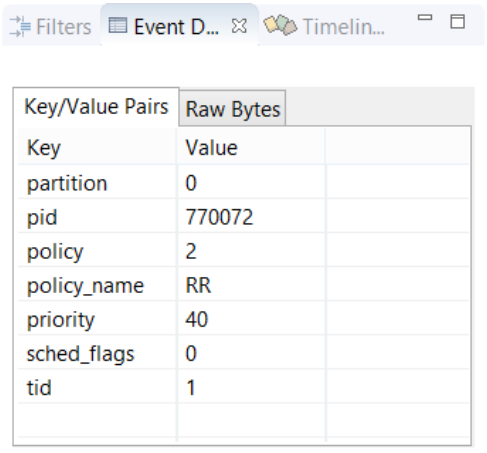
Page updated: August 11, 2025

Event Data

This view displays the data for an event selected in another System Profiler view such as **Trace Event Log** or even the results in **Search**.

The **Event Data** view is shown in the bottom left corner instead of in the area below the editor like most profiling-related views. There are two tabs:

- **Key/Value Pairs** — Lists the names and values of individual event data fields
- **Raw Bytes** — Displays the individual data bytes, in hexadecimal format



In the table, you can select one row (by left-clicking) or many rows (by holding **Ctrl** and **Shift** while clicking) and then copy the data to the clipboard, by right-clicking and choosing **Copy**. This is handy for saving specific event data outside of the IDE.

Page updated: August 11, 2025

Event Owner Statistics

This view displays per-owner event statistics, including event counts and how much time owners spent in specific states.

Every statistics view in the System Profiler perspective has these common behaviors:

- No trace data are displayed until you click the **Refresh statistics** button (); some views have a different name for this button but it does the same thing.

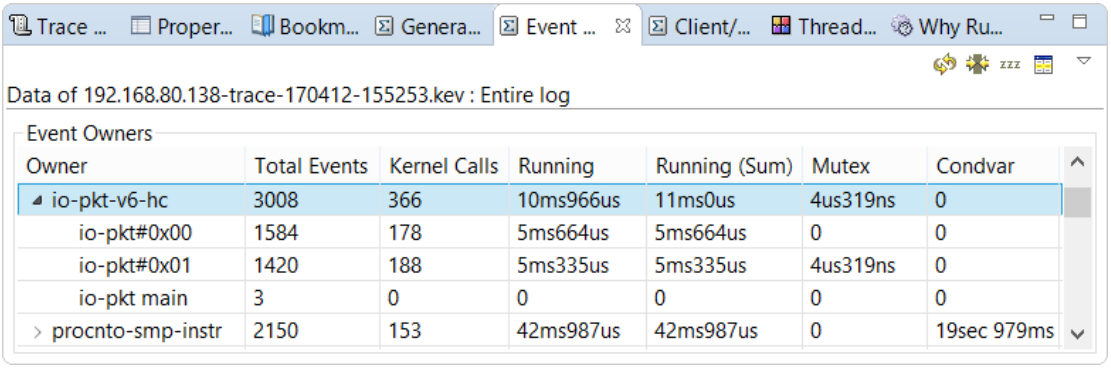
  > **NOTE:**
  > If you're viewing statistics from one kernel event log file but then open another log file, the results in the statistics-related views won't reflect the newly opened file until you refresh them manually.

- They can display statistics that reflect either the entire trace period or the timeframe selected in an editor pane. The display mode is controlled by the **Toggle global/selection statistics gathering** button () as well as the top option in the upper right dropdown ().
- They can generate a CSV report of the data currently displayed. This feature is accessed through the bottom option in the upper right dropdown.

Each row in the **Event Owner Statistics** table shows data for one owner, with its name given in the leftmost column. The total events and number of kernel calls are given in the next two columns, while the remaining ones list the state times for a configurable set of states. By default, the table shows the times spent in these states:

- Running — For processes, the total time within the trace when at least one of their threads was running. The Process & Thread Activity description for the Summary pane explains how this metric is calculated. For threads, their total time executing on a CPU.
- Running (Sum) — For processes, the sum of their thread runtimes. This column can vary from the last one only for processes on multiprocessor or multicore systems. For threads, these two metrics are the same. Again, more details are given in the Process & Thread Activity description.
- Mutex — Total time blocked on a mutual exclusion lock (i.e., inside *pthread_mutex_lock()* calls).
- Condvar — Total time blocked on a condition variable (i.e., inside *pthread_cond_wait()* calls).

Note that you can resort the table by clicking column headers.



Owner	Total Events	Kernel Calls	Running	Running (Sum)	Mutex	Condvar
io-pkt-v6-hc	3008	366	10ms966us	11ms0us	4us319ns	0
io-pkt#0x00	1584	178	5ms664us	5ms664us	0	0
io-pkt#0x01	1420	188	5ms335us	5ms335us	4us319ns	0
io-pkt main	3	0	0	0	0	0
procnto-smp-instr	2150	153	42ms987us	42ms987us	0	19sec 979ms

The view toolbar has two other buttons:

- **Include idle threads** () — When selected, the table data reflect the time spent in idle threads. By default, this is unselected.
- **Synchronize with editor filters** () — When selected, the table shows data from only those owners specified in the **Filters** view. By default, this is unselected.

The view dropdown has two additional options:

- **Flatten event owners** — When selected, no summary rows for interrupts and processes are shown; instead, only rows for interrupt handlers and threads are displayed in a flat list format. This setting is off by default, so initially, you see only interrupt and process entries, which you must expand to see individual handlers and threads. In the previous screenshot, the `io-pkt-v6-hc` process is expanded to reveal the data for its threads.
- **Configure Table** — Opens a window for picking the states for which you want to see time data in the table.

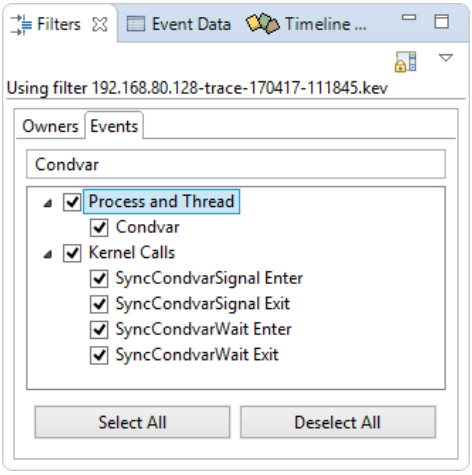
Page updated: August 11, 2025

Filters

Filtering lets you view a subset of the captured information. Using the **Filters** view, you can restrict which owners and events are represented in the Timeline pane and some statistics-related views.

In the **Filters** view, which is shown in the bottom left corner, there are two tabs:

- **Owners** — This left tab lets you check boxes to choose the owners to include in the displayed results. You can expand interrupt and process entries to choose individual handlers and threads. Just above the list is a text field that lets you filter the owners based on matching text; the IDE updates the list as you type in keywords. Below, there are radio buttons for sorting the owners by name or by PID, as well as buttons for selecting or deselecting all list entries.
- **Events** — This right tab lets you select specific event classes or types to include in the results. You must expand class entries to see individual types. Like the first tab, there's a text field for filtering the list based on keywords and buttons for selecting or deselecting all list entries.



NOTE:
Filters, especially event-based ones, can cause some IDE tools to give incorrect answers because they no longer have a complete data set. Generally, you'll want to filter by owner—there aren't many cases when filtering by event is necessary.

This view provides similar controls as the *traceparser*() API, which allows you to parse a log file. A System Profiler filter is different from a kernel filter. This first type of filter applies to Step 4 of the kernel tracing process, so it affects what's shown in the IDE, not what's stored in the log file. Events filtered out at this stage aren't lost, unless you save the filtered data and overwrite the original file.

You can also access some built-in (pre-defined) filters, through the upper right dropdown (▽). The following submenus are available:

- **Select using log** — Add data for specific owners and events to the filtered set
- **Deselect using log** — Remove data for specific owners and events from the filtered set
- **Show only using log** — Replace the filtered set to show only the data for specific owners and events

The right-click menu for the Timeline pane has equivalent submenus, with the different names of **Show more**, **Hide**, and **Show only**. Because this pane is synchronized with the filter settings, you should switch to it before selecting a filter. Each submenu lists these pre-defined filters:

Critical Threads (owners) and Critical Events (events)
: Include event data from threads explicitly assigned to a partition. For the first option, you'll see events related to adaptive partitioning scheduling (APS), state changes, pulses and messages, and more. For the second option, you'll see only Running events. These filters let you find areas of concern in a system with APS.

Selected (owners)
: Include event data for the owners currently selected in the Timeline pane. All events from these owners are added to the filtered set.

State Activity (owners)
: Include event data for all threads that changed state during the trace.

IPC Activity - All (owners) and IPC Activity - Selected Owners (owners)

Include data from threads involved in any form of IPC (e.g., message passing, signalling). The first option filters in data for all such threads while the second one gets data for those selected in the Timeline pane and those they communicated with.

CPU Usage - All (owners) and CPU Usage - Top 10 (owners)

Include data from threads that consumed CPU time. The two options let you get data from all threads or the ten that consumed the most CPU time within the entire system, regardless of which processes they belong to.

When you change the filter settings, the checkboxes in the **Owners** and **Events** tabs are updated to reflect the data now included in the results set. The Timeline pane is also refreshed to show those owners defined by the active filter.

NOTE:

The Timeline pane is the only editor pane affected by the selections in this view; the data shown in other panes can't be filtered. The **Event Owner Statistics** and **Trace Event Log** views can be synchronized with the filters by clicking the **Synchronize with editor filters** button () in their upper right toolbars. The **General Statistics** view is always synchronized.

Finally, the view dropdown has two other options:

- **Switch Filter** – This submenu lists all kernel event log filters available in the workspace. These filters have the same names as the log files. A filter is implicitly created for each kernel event trace that you run from the IDE.

  When you select a filter, the owner and event settings defined by it are applied to the displayed trace results, causing the **Filters** view and Timeline pane to be refreshed.

- **Configure Filters** – This menu option opens a window that lists the available filters. You can click a list entry and then the **Set Active** button to change filters. There are also buttons for creating a new filter, duplicating or renaming it (when one filter is selected), combining multiple filters (when you use multi-select), and deleting filters. Also, you can import and export filters, as XML files.

  At the top of the popup window is a checkbox labelled **Lock active filter**, which controls whether the same filter gets applied to the trace results for all log files currently open. This checkbox is equivalent to the Toggle Display Lock button () shown in the view toolbar. When you enable the locking through one of these UI controls, the other one is immediately updated.

Page updated: August 11, 2025

General Statistics

This view provides event statistics for all displayed owners or the selected ones. The statistics include aggregate data about the times spent in each state and the number of non-state events of each type.

Every statistics view in the System Profiler perspective has these common behaviors:

- No trace data are displayed until you click the **Refresh statistics** button (); some views have a different name for this button but it does the same thing.

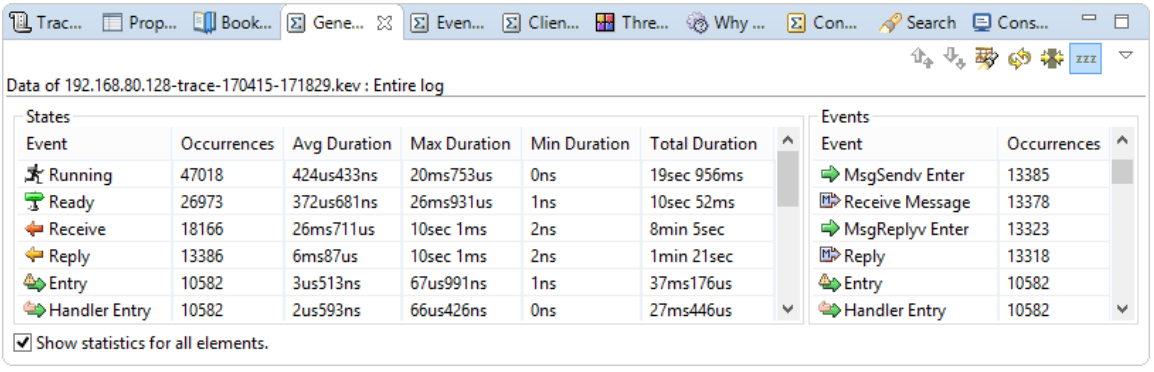
  > **NOTE:**
  > If you're viewing statistics from one kernel event log file but then open another log file, the results in the statistics-related views won't reflect the newly opened file until you refresh them manually.

- They can display statistics that reflect either the entire trace period or the timeframe selected in an editor pane. The display mode is controlled by the **Toggle global/selection statistics gathering** button () as well as the top option in the upper right dropdown ().
- They can generate a CSV report of the data currently displayed. This feature is accessed through the bottom option in the upper right dropdown.

The data shown in **General Statistics** are calculated from only those events belonging to owners currently displayed in the Timeline pane; which owners are displayed depends on any filters you've applied. Unlike with some other System Profiler views, there's no button to turn the filtering on or off for the table data—the data are always synchronized with the filters.

The view contains two tables:

- **States** – This left table provides statistics about state entry events, including the number of occurrences and the average, maximum, minimum, and total duration in each state.
- **Events** – This right table displays the number of occurences for non-state events that marked a new task involving the kernel, such as a message send, mutex locking, or condvar signalling.



Initially, the **Show statistics for all elements** box (at the bottom) is checked, so the table data represent all displayed owners. But you can uncheck the box to see data from only those that are selected, meaning their timelines are highlighted in grey. This option provides a second level of filtering.

In the **States** table, you can right-click in a row and from the context menu, navigate directly to the events that mark the beginning of the maximum and minimum times in that state. In the Timeline pane, the IDE moves the yellow arrow marker to the matching event's tick mark and draws thin red circles around this tick mark and those of all other events of the same type, for all displayed owners. Note that there are equivalent buttons, **Go to maximum duration** () and **Go to minimum duration** () in the view toolbar.

After selecting a row in either table, you can click the **Search for selected events** button () to search for all events of the same type within the displayed owners. The IDE then switches to the Search view, which lists the results up to a maximum of 1000 entries. Note that the **Events** table makes this same feature available through the right-click menu.

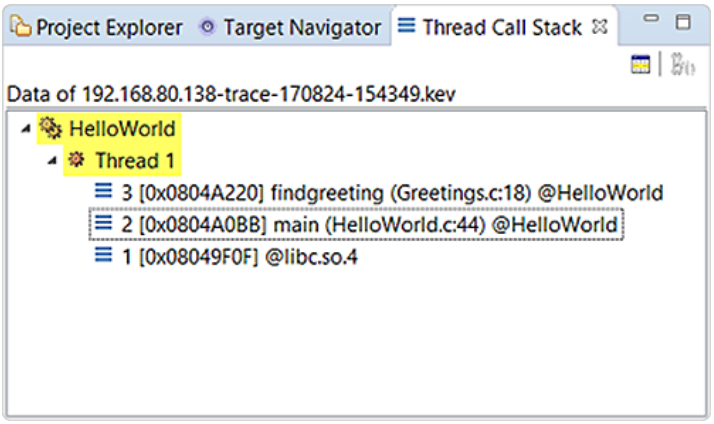
Finally, the toolbar has one more button: **Include idle threads** (). When this option is selected, the table data reflect the time spent in idle threads. By default, this is unselected.

Page updated: August 11, 2025

Thread Call Stack

In this view, you can examine the thread call stacks for a particular time point in the trace.

NOTE:
Only threads from processes with instrumentation code appear in the **Thread Call Stack** view. To see call stack data for an application, you must build it with function instrumentation. This way, the profiler library (**libprofilingS.a**) records all Function Enter and Function Exit events, and uses them to determine the stack at any point in execution.



Examining call stacks is useful for seeing when an application's functions were executed relative to other system events. When you click an event in the Timeline pane or the **Trace Event Log** view, the IDE populates the **Thread Call Stack** view with the call sequences for all instrumented threads that were active at the time of the selected event. You can double-click a call stack entry to open the associated source file at the corresponding line of code.

If you don't see source file and function names and line numbers in the stack entries, it means address translation isn't properly configured. Information about enabling and using this trace feature is given in the "Address Translation" subsection of the kernel log configuration reference. When address translation is disabled, the view shows only virtual addresses.

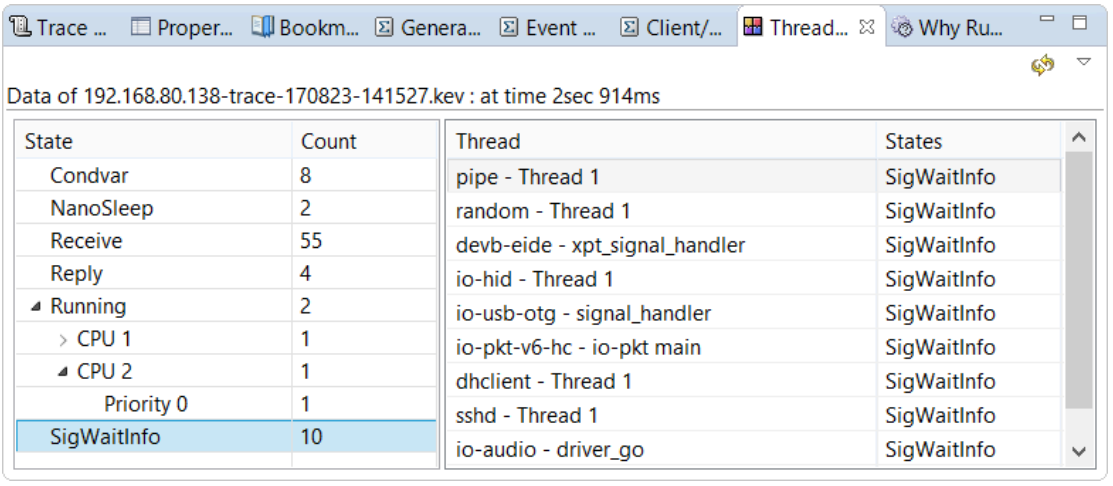
In the view toolbar, there are two buttons:

- **Synchronize with editor filters** ( ) — Updates the data display based on the current filtering settings.
- **Export to application profiler session** ( ) — Takes the data from the open kernel event log (**.kev**) file and exports it to an Application Profiler session.

Page updated: August 11, 2025

Thread State Snapshot

This view shows you the states that various threads were in at a given point during the trace. This information helps you understand what the system was doing at that time.



The contents of the view are based on the current selection in the Timeline pane. After you click an event in this editor pane, you must then click the **Refresh** icon () in the **Thread State Snapshot** view's toolbar to update the contents of this view.

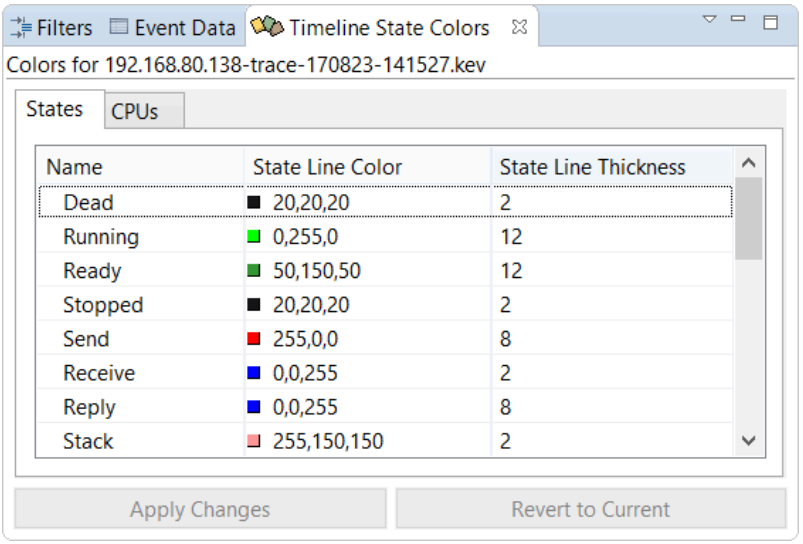
Two tables are shown. The first table lists the thread counts for all states. For the Running state, you can click the arrow next to its label to see the thread counts for individual CPUs on the target system. You can then click the arrow next to a CPU label to see the thread counts for specific priority levels. Meanwhile, the second table lists the names of the threads in the states currently selected in the first table. Note that you can use multi-select in the first table to list the threads for multiple states.

By default, the tables are stacked horizontally, but you can select **Vertical View Orientation** from the view's dropdown menu () to stack them vertically, with the state count table appearing above the thread names table. To restore the default display, choose **Horizontal View Orientation** from the dropdown menu.

Page updated: August 11, 2025

Timeline State Colors

If you want to change the color settings used by the Timeline pane to represent thread states, you can define a new coloring scheme in the **Timeline State Colors** view.



NOTE:
With the default System Profiler perspective layout, this view is shown in the bottom left area. You can explicitly open it by selecting **Window** > (and then)**Show View** > (and then)**Other** > (and then)**QNX System Profiler** > (and then)**Timeline State Colors**.

The view displays two tabs:

- **States** — Controls the color and thickness of the lines used to represent specific states on a thread's timeline. Click a second or third column entry to change its value.
- **CPUs** — Controls whether a line is drawn for the `procnto` idle thread running on a particular CPU and if so, the color of the line. Click a third or fourth column entry to change its value.

The default settings generally assign the same color to multiple states related to similar activities. For instance, Send and NetSend are given the same color, as are CondVar and Semaphore.

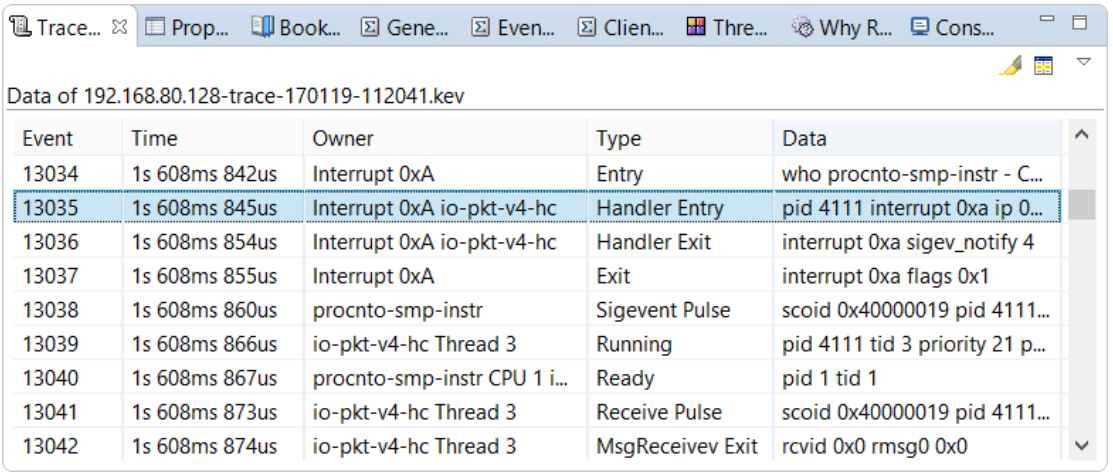
You can apply any new settings you've made so far by clicking **Apply Changes**. The IDE prompts you to save them and if you say yes, it opens a file selector. The exported color settings are encoded in XML format, so you should give your output file a **.xml** extension. Note that you can also save the settings by clicking the view's dropdown menu (▽) and choosing **Export**, as well as import previously saved settings by choosing **Import**.

The **Revert to Current** button restores the color settings to their state when you last applied the changes or imported settings.

Page updated: August 11, 2025

Trace Event Log

This view displays details for the events surrounding the event selected in the Timeline pane.



Event	Time	Owner	Type	Data
13034	1s 608ms 842us	Interrupt 0xA	Entry	who procnto-smp-instr - C...
13035	1s 608ms 845us	Interrupt 0xA io-pkt-v4-hc	Handler Entry	pid 4111 interrupt 0xa ip 0...
13036	1s 608ms 854us	Interrupt 0xA io-pkt-v4-hc	Handler Exit	interrupt 0xa sigev_notify 4
13037	1s 608ms 855us	Interrupt 0xA	Exit	interrupt 0xa flags 0x1
13038	1s 608ms 860us	procnto-smp-instr	Sigevent Pulse	scoid 0x40000019 pid 4111...
13039	1s 608ms 866us	io-pkt-v4-hc Thread 3	Running	pid 4111 tid 3 priority 21 p...
13040	1s 608ms 867us	procnto-smp-instr CPU 1 i...	Ready	pid 1 tid 1
13041	1s 608ms 873us	io-pkt-v4-hc Thread 3	Receive Pulse	scoid 0x40000019 pid 4111...
13042	1s 608ms 874us	io-pkt-v4-hc Thread 3	MsgReceivev Exit	rcvid 0x0 rmsg0 0x0

With this view, you can see every kernel event that occurred during the trace period and the exact ordering of the events. Given the large number of events (even in relatively short traces), you may want to show only certain events, to find areas of interest. You can click the **Enable/disable event filtering** button ( ) in the view's toolbar to toggle between showing all events or only those that match the filters defined in the Filters view. Or, you can show all events and highlight the ones that match, by clicking the highlight button ( ).

Events are listed in table format, in chronological order. The default columns are:

- Event — A unique number indicating the event's relative occurrence within the trace period.
- Time — The event's timestamp, relative to the trace start.
- Owner — The event owner, which can be a process, thread, interrupt vector, or interrupt handler.
- Type — The event type.
- Data — Data associated with the event. Often, the PID and TID are included; for message-related events, the message ID is included.

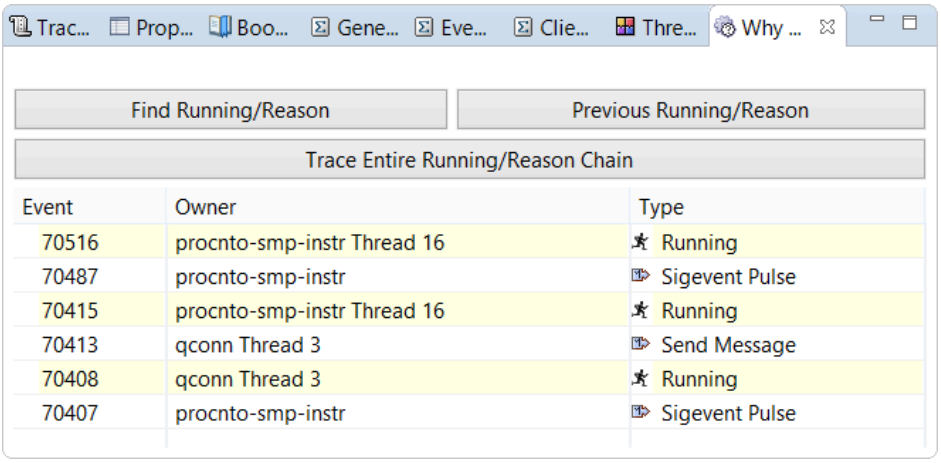
You can display additional columns or change the order of the columns. The view's dropdown menu ( ) contains one option, **Configure Trace Event Table**, which opens a popup window in which you can configure the table display.

As you navigate events in the timelines by using the System Profiler toolbar buttons, the selection in **Trace Event Log** gets updated and the view scrolls to the new locations.

Page updated: August 11, 2025

Why Running?

This view works with the Timeline pane to provide developers with a single-click answer to the question: “Why is this thread running?” In this context, “this thread” refers to the owner of the currently selected event.



First, you must click an event tick mark in the Timeline pane. The editor highlights the owner's name and draws an arrow pointing downwards, just above the event. Then, you can click **Find Running/Reason** in the **Why Running?** view to populate it with event information showing why the current thread could run. Two events are listed:

- The Running event for the current thread
- The event that made the previously running thread change state, allowing the current thread to run

To learn why the previously running thread could run earlier, click **Previous Running/Reason** to see the two events related to that thread's transition to the Running state. You can repeat this action to get a clear view of the sequence of activities that led to the original execution position. Or, you can click **Trace Entire Running/Reason Chain** (either just after you clicked an event tick mark or any time when you see the partial execution sequence) to display the entire backtrace of Running events.

If you click a tick mark for a `procnto` idle thread or the **Previous Running/Reason** button when the entire backtrace is already displayed, you'll get a message saying the end of the chain was detected, and no new events will be listed.

NOTE:
Don't confuse the Running backtraces shown here with the call stacks shown by the **Memory Backtrace** view when you're debugging memory problems. The Running backtraces reveal the cause/effect relationship between threads that led to current thread's execution whereas the memory backtraces show the function call sequence for only the current thread's execution position.

Page updated: August 11, 2025

System Profiler editor

When you open a kernel event log file, whether you're prompted by the IDE after a kernel event trace finishes or you manually open a file copied into your workspace, the IDE uses the System Profiler editor to visually present the file's data.

NOTE:
Due to the large amount of data generated by kernel tracing, the IDE can't display data while tracing is in progress. The System Profiler editor is used to show kernel event details gathered during the trace only after it's finished.

The editor lets you examine the captured events through numerous panes, which visualize event data in different ways. Initially, the editor displays the Summary pane, which shows general trace statistics such as CPU time breakdown. But you can open another pane by clicking the Switch Pane dropdown in the upper right controls ( ) and selecting an option. Or, you can right-click in the editor and select **Display** > (and then)**Switch Pane** and then a particular pane.

The System Profiler editor panes include:

- Summary
- CPU Activity
- CPU Migration
- CPU Usage
- Inter CPU Communication
- Partition Summary
- Timeline

You can display more than one pane at a time by using the Split Display dropdown ( ). This is useful for looking at different sections of a log file and doing comparative analysis. When you click the name of a pane that's not currently displayed, the IDE opens it. You can open up to four panes, although it can be hard to read statistics when that many are displayed. The right-click menu has an equivalent submenu, **Display** > (and then)**Split Display**, as well as the **Display** > (and then)**Switch Orientation** option, which lets you alternate between stacking the panes horizontally or vertically.

With the split display, any newly opened pane displays data based on the same events selected in the previously opened one. But the pane becomes independent right away, meaning you can change the zoom level or highlight a timeframe in its display without affecting the events selected in other panes.

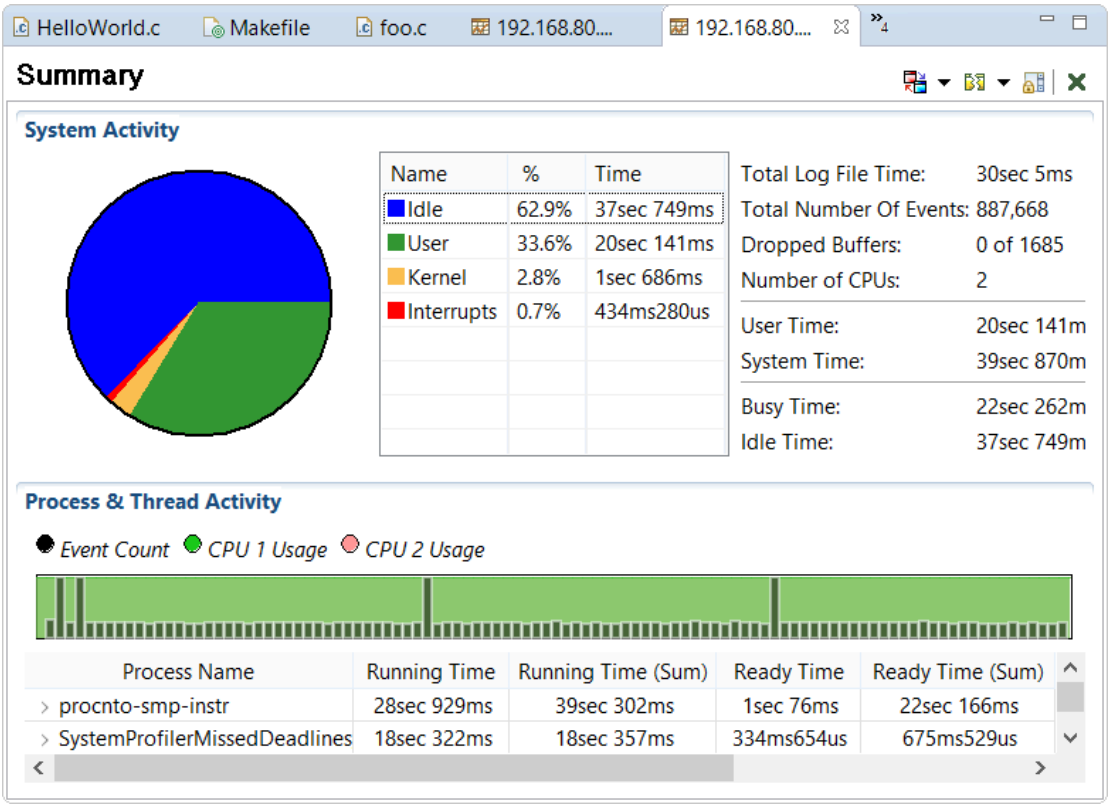
If you want to synchronize the selection in multiple panes, click the Toggle Display Lock button ( ) in each one, or choose **Display** > (and then)**Lock Pane** from the right-click menu; for convenience, this menu also has the **Display** > (and then)**Lock All Panes** option. After you do this, any timeframe selection or zooming that you do in one pane automatically updates the display in the other locked panes.

To close a pane, click the close button ( ) or choose **Display** > (and then)**Close Pane**. You must leave one pane open, so this button and option do nothing when a single pane is displayed.

Finally, the right-click menu has the **Open with QNX Application Profiler** option, for extracting any function profiling data into a new profiling session. Note that the **Bookmark** option, although present in the context menu of any pane, really applies to the Timeline pane only.

Summary

This pane shows a summary of the kernel, process, and thread activity that occurred during the trace.



In the upper left corner, the piechart and table show you how busy the target machine was over the trace period. This graph and its accompanying statistics reveal the CPU time spent being idle, running user code, running kernel (system) code, or processing interrupts. In the upper right corner, the kernel event trace statistics include but aren't limited to:

- how long the trace ran
- how many events occurred
- user time, which is the CPU time spent running user code
- system time, which is the sum of the idle, kernel, and interrupts times

The Process & Thread Activity area contains a bar graph illustrating the change in CPU usage and the event rate. Below the chart, a table lists the processes and threads and for each one, reveals the time spent in the Ready, Running, or any blocked state, and the number of kernel calls made and messages sent. For each state, two columns are shown:

***State* Time**

For processes, this is the total time during the trace period when at least one of their threads was in the given state.

Consider a process with two threads running on different processors or cores. The first thread runs from 0 to 3 s within the trace period and the second thread from 0 to 1 s. The **Running Time** column shows 3 seconds because the two threads ran over a three-second interval.

For threads, which can run on only one CPU core at a time, this is their total time in that state throughout the trace.

***State* Time (Sum)**

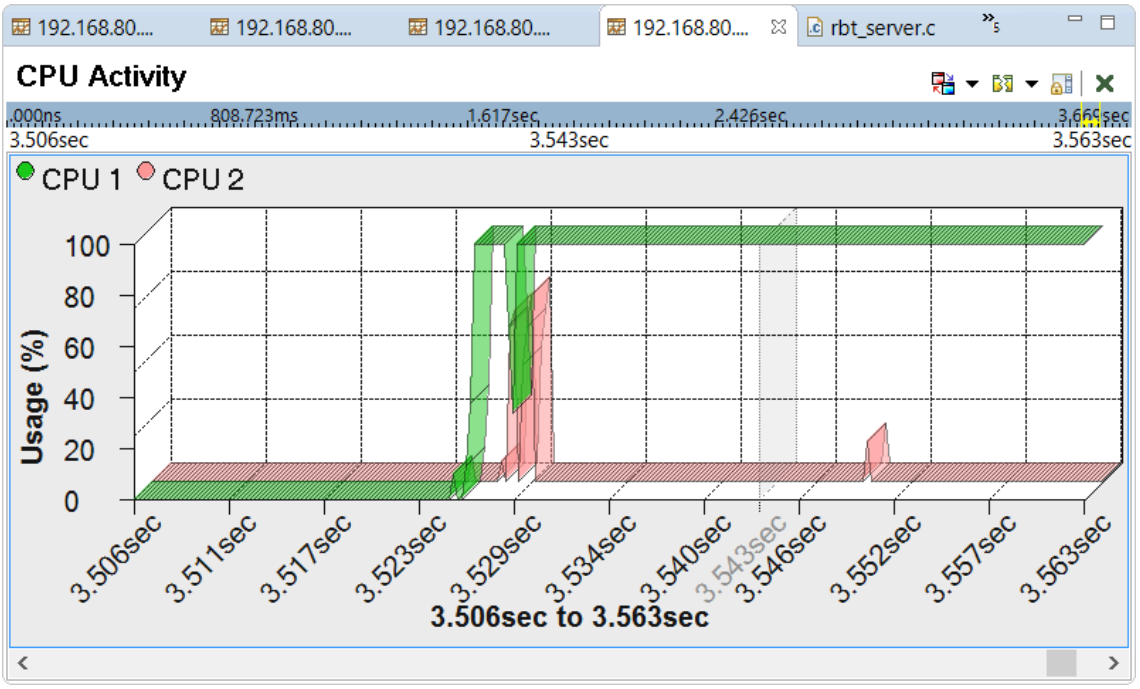
For processes, this is the sum of all individual thread times for that state, throughout the trace. In the previous example, the **Running Time (Sum)** column shows 4 seconds because this is how much runtime was used by the two threads, even though their overlapping execution meant they finished running by the 3 second mark.

> **NOTE:**
> This column can differ from the previous one only for processes on multiprocessor or multicore systems. With single-processor single-core systems, only one thread in the system can be moved into a given state at a time. For threads, the columns always match because they can't run on multiple CPU cores concurrently.

The Process & Thread Activity table contains the same information that you can extract by examining a particular time range of the kernel event log using the General Statistics view.

CPU Activity

This pane displays how the load changed on each CPU over time, as a percentage of cycles consumed.



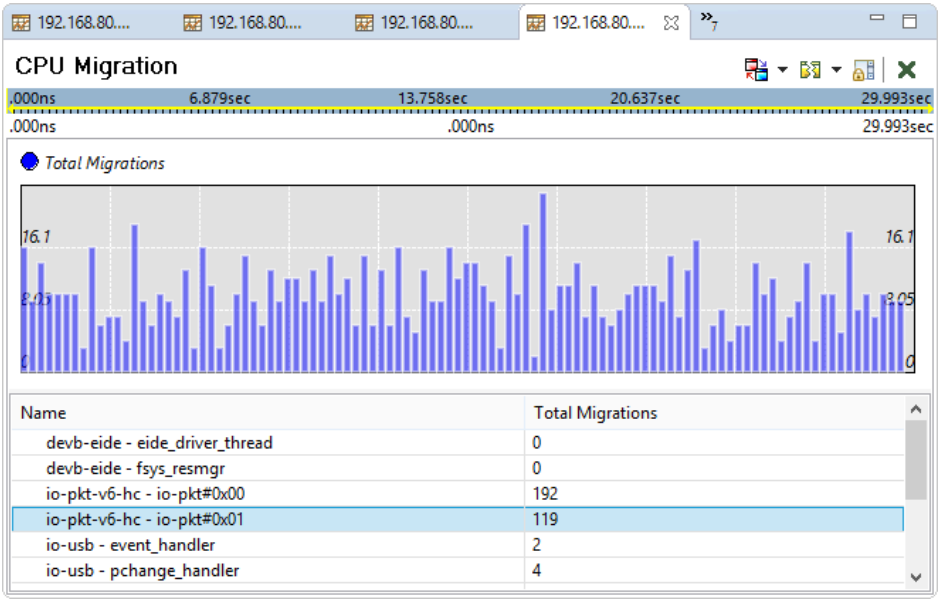
The default chart type is a 3D line graph. You can right-click anywhere in the pane and in the **Graph Type** submenu, select another type based on a 2D or 3D display, lines or bars, or what shading to use.

The zoom feature lets you scale the graph to better see changes in CPU load over short timeframes. To change the zoom level, you can use the zoom in (🔍) and zoom out (🔍) buttons in the upper left corner of the IDE. There are equivalent options in the right-click menu. For **Zoom In**, you can first select a region of the graph by clicking and dragging the mouse, then click this option to zoom in on the highlighted timeframe. The right-click menu also has the **Zoom Level** submenu, which lets you return to the full-scale graph (i.e., at 100%) or set a custom zoom level. Above the graph, the yellow band in the timescale and the start and end times are updated to reflect the time period now displayed.

CPU Migration

The CPU migration pane tells you how many times the kernel migrated threads between CPUs, for whatever reason. This information is useful for finding performance problems on multiprocessor or multicore target systems. Note that this pane is meaningful only for these system types.

The bar graph in the top area illustrates the number of thread migrations over time. The table in the bottom area lists the total migrations for each thread that was active during the displayed timeframe; this table provides data for all threads within processes as well as interrupt handlers.



The zoom feature lets you scale the graph to better examine CPU migrations over short timeframes. The controls are the same as those for the CPU Activity pane. Unlike in this other pane, however, you can't change the chart type.

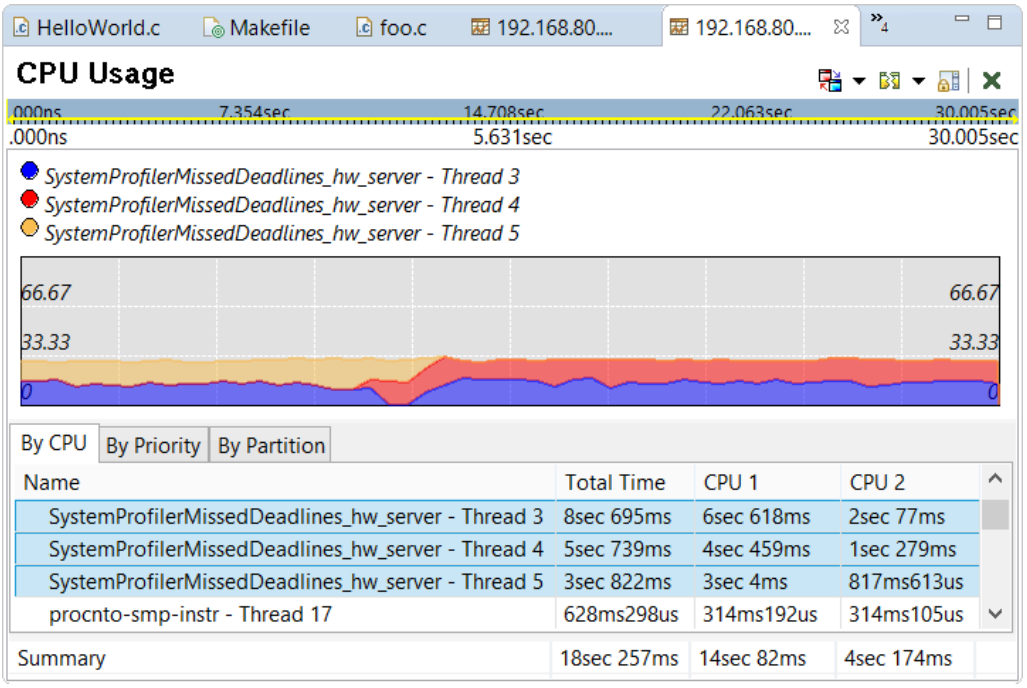
Individual bars represent the number of CPU scheduling migrations over fixed time intervals. The migration count is incremented whenever a thread switches CPUs, so the peaks in the graph indicate times when there was a lot of contention for certain CPUs. Frequent CPU migration reduces performance because the instruction cache is regularly flushed, invalidated, and reloaded on the new CPU.

The length of the time intervals represented by the bars is based on the time period displayed. Thus, when you zoom in, these intervals get shorter. As well, the vertical axis of the graph is rescaled based on the maximum number of migrations for a thread in the newly selected timeframe, and the table is refreshed based on the graph.

In addition to total migrations, the kernel trace also tracks how many threads are migrated due to cross-CPU communication. This is shown in Inter CPU Communication.

CPU Usage

This pane shows the CPU usage history for various event owners. CPU usage is amount of runtime that threads get. The graph draws separate colored lines and shaded areas for each owner selected in the table underneath. In this example, we selected process threads, but the functionality works the same for interrupt handlers. For ISTs, select the thread attached to the IST.



The zoom feature lets you scale the graph to better read changes in CPU consumption over short timeframes. The controls are the same as those for the CPU Activity pane.

The table area contains three tabs that display the times that various threads ran on individual CPUs, at specific priority levels, and within certain partitions. The last tab shows data only if the target uses adaptive partioning. For all three tabs, the threads are initially sorted by their total running times across all CPUs (in descending order), but you can click the column headers to change the sorting order.

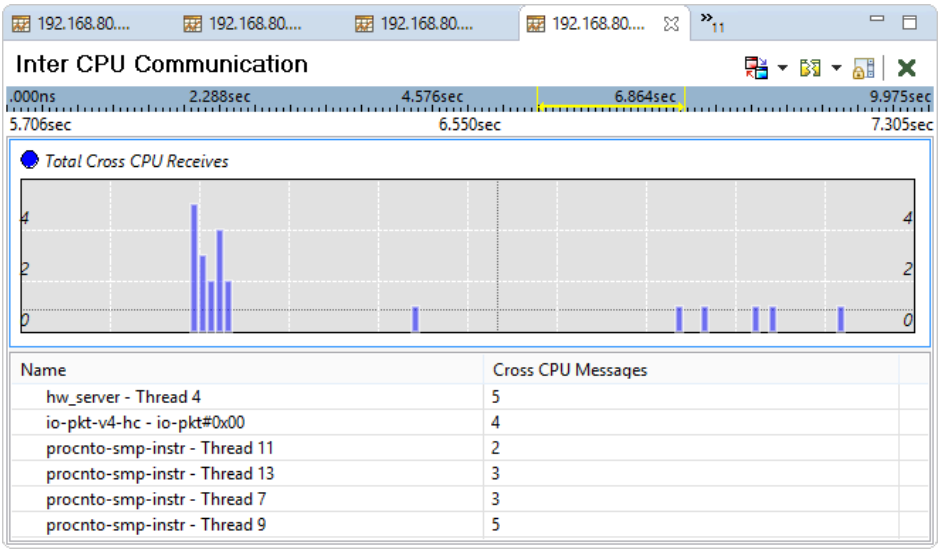
The measurements reflect the CPU consumption over the time period selected in the chart. By default, they appear as percentages followed by time values, but you can display just one of these metrics or both of them in reverse order, by selecting **Window** > (and then)**Preferences** > (and then)**QNX** > (and then)**System Profiler** > (and then)**CPU Usage** and clicking an option in the **Table Data Format** dropdown. This same preference dialog lets you switch between a line and an area chart, using the **Chart Type** dropdown.

Clicking table rows updates the graph to display the CPU usage history of the associated threads. Multi-select is supported, by holding the **Ctrl** and/or **Shift** keys, similar to Windows Explorer. Below the table, a summary row displays the sums of the values for the selected threads.

Inter CPU Communication

The Inter CPU Communication pane shows you how often the kernel migrated threads due to message passing. This information helps you find performance bottlenecks caused by messaging on multiprocessor or multicore target systems. Note that this pane is meaningful only for these system types.

In the top area, the bar graph illustrates the number of thread migrations over time. In the bottom area, the table lists the total migrations for each thread that was active during the displayed timeframe.



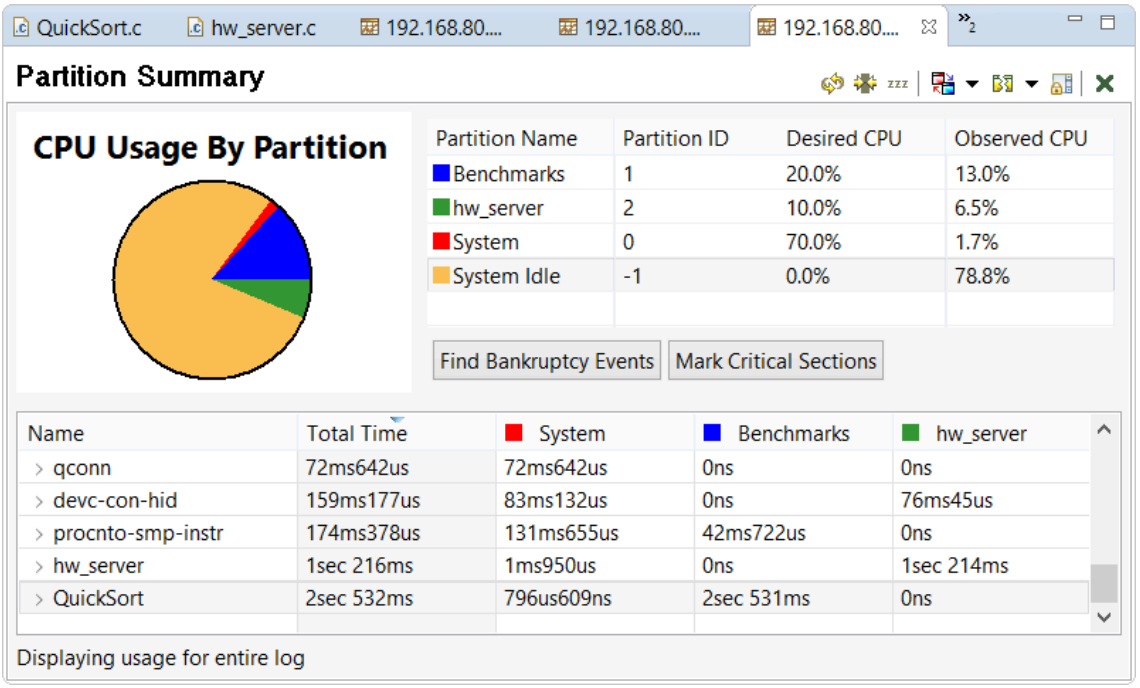
This pane tells you how many cross-CPU messages were exchanged. When a message-sending (client) thread and the corresponding message-receiving (server) thread run on different CPUs, the message sending data must be migrated from the client's CPU to the server's. This activity reduces performance because the first CPU's cache is invalidated when the data get moved.

You can adjust the zoom level to see the message-related migration counts for specific time periods. The bars then change to reflect new fixed time intervals based on the timeframe now displayed and the table gets updated, similar to the CPU Migration pane. Note that this other pane reports how often threads were migrated for *any* reason, not just due to messaging.

Partition Summary

This pane displays the CPU usage of partitions and individual event owners that ran within each one.

> **NOTE:**
> QNX OS 8.0 does not support adaptive partitioning; if your target is based on 8.0, then the IDE won't receive any APS data and this pane isn't meaningful. For targets based on an earlier OS version, this pane is meaningful only when the target uses adaptive partitioning.



In the top left corner, the piechart illustrates the breakdown of CPU usage by partition. Next to it, the table lists the partitions and their desired CPU load or *budget* (what percentage of cycles they're guaranteed in a fully loaded system) and observed CPU load (what percentage of cycles they got). Just below the table, there are two buttons:

- **Find Bankruptcy Events** — searches for all APS Bankruptcy events and displays the results in the **Search** view

- **Mark Critical Sections** — adds bookmarks for all events marking the entry into a critical section; the newly marked events are listed in the **Bookmarks** view

In the bottom area, the table lists the per-event owner CPU usage across all partitions and within each one. A process can run threads in separate partitions, and a thread can switch partitions. For more information about adaptive partitioning and the API used to configure it, see the *Adaptive Partitioning User's Guide* and the *SchedCtl()* reference for the OS version that your target is based on.

NOTE:
To see realtime statistics about the current load, you can examine the APS View in the System Information.

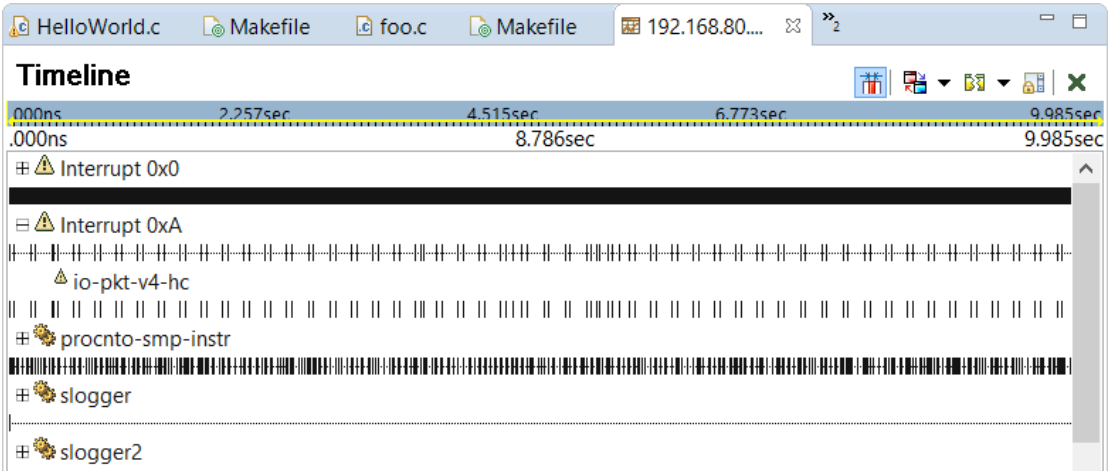
By default, the statistics shown reflect the entire trace period, even if you selected a time range in another pane. To see the CPU usage metrics for only the highlighted range, click the **Focus on Range Selection** button (), then click **Refresh** ().

Whenever you select another time range, you must refresh this pane using the second button. You can click the two buttons again (in the same order) to disable the range selection focus and redisplay the statistics from the entire kernel trace. The reason for this design is that the calculations for finding CPU usage by partition are intensive enough that the System Profiler doesn't automatically perform them as soon as you change the timeframe selection.

At the bottom of the pane, the status bar indicates the time range that the current statistics reflect.

Timeline

This pane provides a detailed chronology of the logged kernel events by illustrating the timing of events belonging to particular owners. The timescale along the top shows the relative start and end times of the trace and highlights the currently selected timeframe. In the main area, timelines are drawn for individual owners, with indvidual events represented by vertical tick marks.



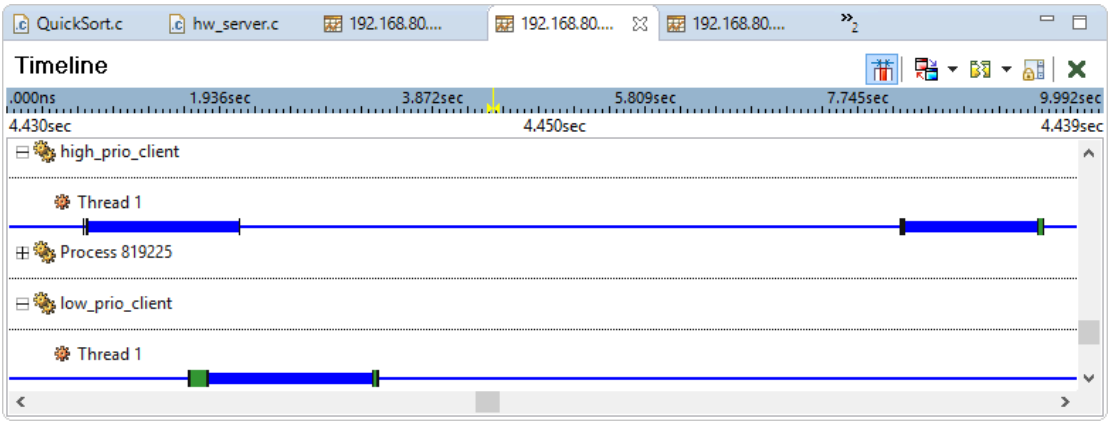
Given the huge number of events in even brief traces, you typically have to zoom in considerably to find an exact event sequence of interest (e.g., a message exchange or unexpected thread scheduling). To change the zoom level, you can use the zoom in () and zoom out () buttons in the upper left corner of the IDE. There are equivalent options in the right-click menu. For **Zoom In**, you can first select a region of the graph by clicking and dragging the mouse, then click this option to zoom in on the highlighted timeframe. The right-click menu also has the **Zoom Level** submenu, which lets you return to the full-scale graph (i.e., at 100%) or set a custom zoom level. Above the graph, the yellow band in the timescale and the start and end times are updated to reflect the time period now displayed.

The Timeline pane lets you scroll vertically, to see timelines from different owners, and horizontally, to navigate to earlier or later in the trace period when you're zoomed in. With this last action, the timescale selection is automatically updated.

NOTE:
Several other panes—CPU Activity, CPU Migration, CPU Usage, and Inter CPU Communication—also display the timescale. When you change the selection in any of these panes or in Timeline, if you then switch to another of these panes, the graph and data tables will reflect the new selection. But when multiple panes are displayed concurrently (using the Split Display feature), changing the selection in one does *not* update the others.

Initially, a summary timeline is drawn for each process and interrupt vector. The tick marks on this timeline represent specific events belonging to any thread within the process or any interrupt events attached to the vector. You can click the plus sign (+) next to the owner's name to display the timelines for its threads or handlers; when you do this, the summary tick marks are cleared. Clicking the minus sign (-) next to an expanded owner hides these other timelines and redraws the summary marks.

The System Profiler illustrates thread states by drawing, on thread timelines, color bands between state change events. Areas with a thinner line represent timeframes when the thread was blocked.



The Timeline pane offers many helpful features for finding key event information:

- Hover-based information
- Owner and event selection
- Navigation
- Scrolling
- Find
- Search
- Bookmarks
- IPC representation
- Event labels
- Renaming processes and threads

Timeline: Hover-based information

When you hover the pointer over an owner name or an event tick mark, the editor shows related information in a popup box. For owners, the details shown depend on the owner type:

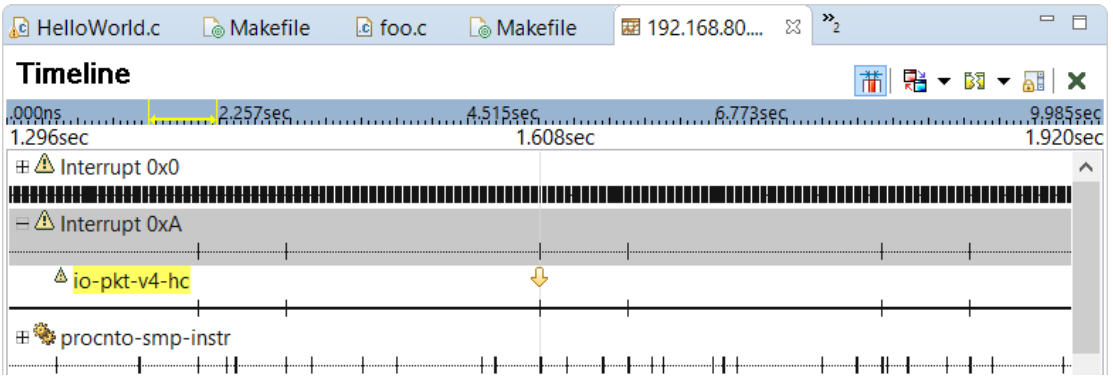
- processes — process name and PID
- threads — TID and parent process information
- interrupts — interrupt vector number, in hexadecimal and decimal
- interrupt handlers — name of process in which handler runs, and interrupt vector number

For events, the details vary with the event type. Generally, you see the owner, event type, event number within the log, timestamp, PID, and other specific data such as the new state or the semaphore address.

Timeline: Owner and event selection

To select an owner, click its name. The owner and its summary timeline are highlighted in grey, and you can then find specific events or navigate between events within the selection. Multi-select is supported, by holding the **Ctrl** and/or **Shift** keys when selecting processes, similar to Windows Explorer.

To select an event, click its tick mark. The editor highlights the owner's name in yellow and draws a marker (which is a yellow arrow pointing downwards) just above the event, as well as a dotted vertical line that lets you see when the event occurred relative to others. Near the top, just below the timescale, the middle value is updated to the selected event's timestamp. Highlighting one event is useful for navigating through other events.



Multi-select is supported for events differently than for owners. You can hold **Shift** and click two events to select the timeframe between them, which is useful for zooming in on a specific event sequence.

Timeline: Navigation

The System Profiler provides toolbar buttons (in the upper left corner) and shortcut keys to navigate between events in the Timeline pane:

To navigate to the:	Click this toolbar button:	Use this shortcut key:
Previous event by the same owner		
Previous event in the selection		**CtrlShiftLeft**
Previous event		**CtrlLeft**
Next event		**CtrlRight**
Next event in the selection		**CtrlShiftRight**
Next event by the same owner		

You can navigate directly to an event. When the editor is in focus, pressing **CtrlL** or selecting **Navigate** > (and then)**Go To Event** opens a window that lets you enter a sequence number, timestamp (in nanoseconds), or event cycle. When you provide one such value and click **OK**, the System Profiler looks for the event. If it finds the event, it moves the yellow arrow marker and highlights the owner in the Timeline pane. For the timestamp, the IDE navigates to the event closest to the specified time.

Some QNX System Profiler views, such as **Trace Event Log** and **General Statistics**, are synchronized with the Timeline pane; selecting an event in one display navigates to that event in the other display. Thus, if the event marker seems to have jumped unexpectedly, most likely, you clicked a table row in one of these views. When you change the timerange selection in the Timeline pane, you can then filter the statistics shown in **General Statistics**; for details, see the General Statistics reference.

Timeline: Scrolling

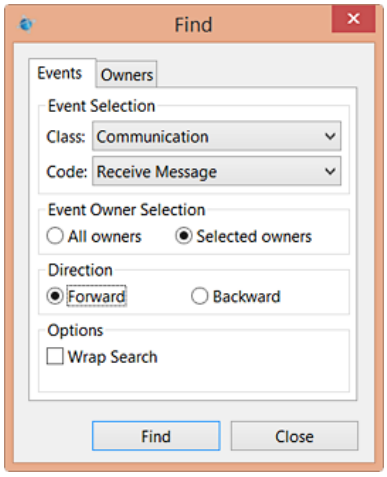
You can move backwards or forwards along the timelines by pressing the left and right arrow keys when the editor is in focus. You can also scroll between event owners:

To navigate:	Use this shortcut key:
To the top of the owner list	**Home**
Up by one screenful	**Page Up**
Up by one owner	**Up**
Down by one owner	**Down**
Down by one screenful	**Page Down**
To the bottom of the owner list	**End**

NOTE:
Scrolling doesn't change the owner or event selection.

Timeline: Find

You search for events by pressing **CtrlF**, selecting **Edit** > (and then)**Find**, or clicking the Find button ( ) in the upper left toolbar. This action opens a window with controls for finding events in open log files:



The **Events** tab, which is initially selected, lets you specify the:

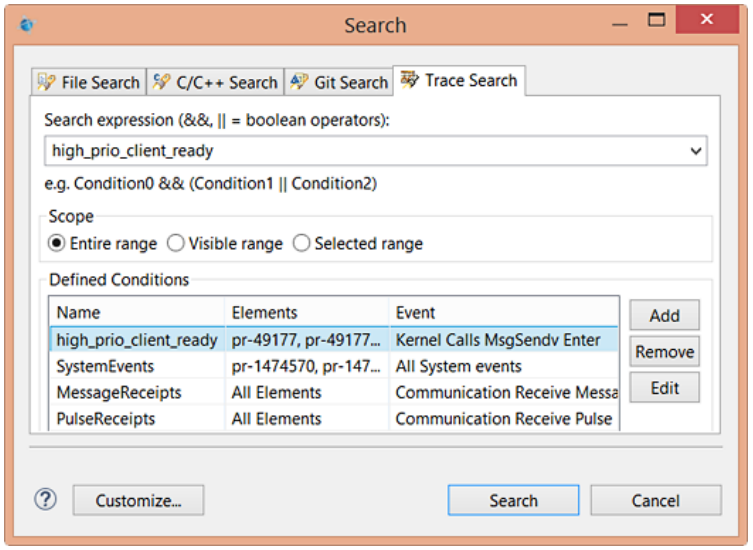
- class
- code (event type)
- search scope, either all owners or the selected ones
- search direction (forward or backward)
- whether to wrap the search

If you click the **Owners** tab on the right, you can then select a single thread in which to search. The text field at the top lets you begin typing the thread name, to filter the list. Multi-select is *not* supported.

When you click **Find**, the System Profiler navigates to the first matching event found. This simple search feature is good for most times when you want to locate an event, but you can also search for events matching specific criteria, as explained below.

Timeline: Search

If you want to see all kernel events matching certain conditions, you can use the search feature. To use it, press **CtrlH** or select **Search** > (and then)**Search**, then click the **Trace Search** tab. The fields for selecting search conditions for kernel events then appear:

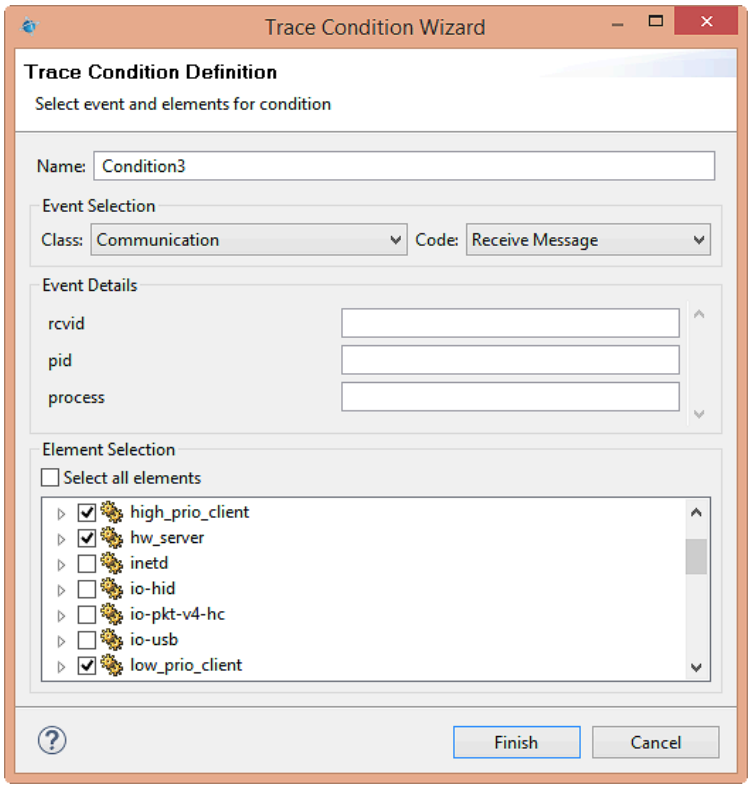


The **Search expression** field at the top lets you name one or more search conditions to use. You can combine results from multiple conditions by specifying the AND (&&) and OR (| |) operators between condition names. You can also use the dropdown on the right to choose exactly one condition.

The radio buttons in the middle let you choose the search scope, which can be the entire trace, the timeframe scrolled to in the editor, or the selected timeframe (which is the area colored light grey).

The **Defined Conditions** panel lists the existing search conditions. On the right are buttons for adding, removing, and editing conditions. The **Add** button opens a wizard that lets you define a new search condition. Clicking the

Edit button after you've selected a list entry opens the same window but any settings made modify the existing condition.



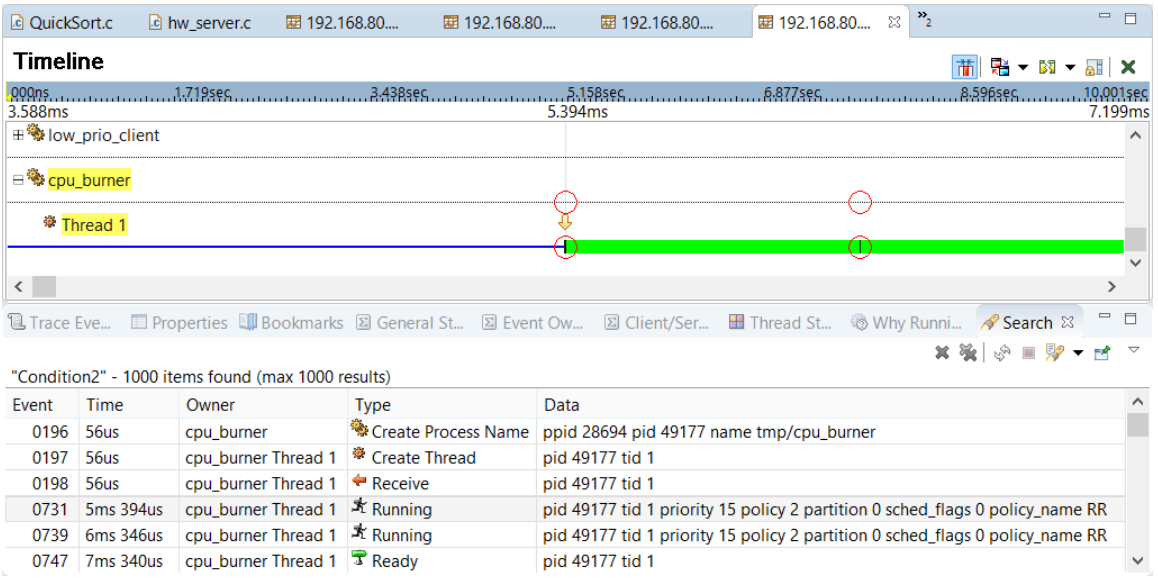
Through the text field at the top, you can give the new condition an insightful name. Below, you can select a class, code (event type), and other data fields related to the code; these last fields appear after you select a specific class and code. By default, the **Select all elements** box is checked (in the bottom area), so the search applies to all owners. But you can uncheck this box and select one or more owners from the list that appears, to restrict the search. Also, you can expand entries for processes and interrupt vectors to choose individual threads and handlers.

Click **Finish** to close the wizard and save the new condition. In the **Trace Search** tab, this condition now appears in the list. When you click **Search** in this tab, the IDE runs the search and lists the results in the **Search** view, with a maximum of 1000 results. Each row lists the following details for an event:

- sequence number within the log
- timestamp, relative to the trace start
- owner
- event type
- all type-specific data fields

You can customize which details are shown by selecting **Configure Trace Event Table** in the upper right dropdown (▽); this action opens a window in which you can select individual columns to display.

If you double-click a row, the System Profiler navigates to that event in the Timeline pane. The owner is highlighted in yellow, the yellow arrow marker is placed above the event's tick mark, and thin red circles are drawn around this tick mark and those for all other events of the same type.



You can select individual rows with the left mouse button and multiple rows by also holding down **Ctrl** and/or **Shift**. This allows you to remove only those events by using the first control outlined just below, but also to copy the corresponding event data to the clipboard (using **CtrlC**) and then paste the data into an Excel workbook or another program that reads data in CSV format.

The upper right toolbar in the **Search** view has the following controls:

- **Remove Selected Items** ()
- **Remove All Items** ()
- **Run the Current Search Again** ()
- **Cancel Current Search** () — This is active only when the search is running.
- **Show Previous Searches** () — This dropdown lists recent searches so you can easily rerun them (by clicking their entries) and provides options to access the full history and clear the history.
- **Pin the Search View** () — This button lets you keep the current results for reference. When this button is clicked, the next time you run a search, the IDE displays the results in a separate window.

Timeline: Bookmarks

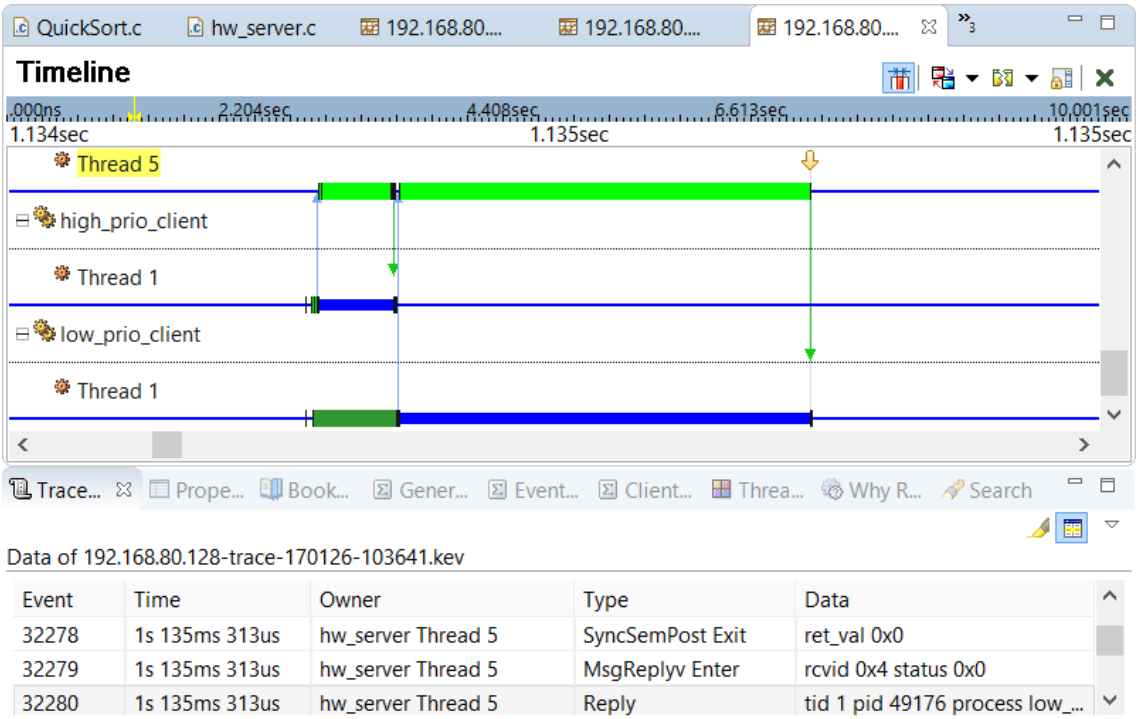
Bookmarks help you keep your place in a kernel event log. To bookmark an event, click its tick mark and then right-click and choose **Bookmark**. (There's also a bookmark button, , in the upper left toolbar.) In the resulting window, you must enter a description for the bookmark. When you do so and click **OK**, the IDE adds the event to those listed in the **Bookmarks** view. In this way, you can annotate a log file to mark the most interesting events.

At any time, you can double-click a row in this view to directly navigate to an event that you bookmarked. You can also choose **Go To** from the context (right-click) menu. This menu also has options for copying the event data to the clipboard, deleting the selected entries, selecting all events, and viewing the properties of a single event. When you select two events, you can choose **Select Range in Editor** to select the timeframe between them.

Timeline: IPC representation

The Timeline pane can illustrate IPC activity such as message exchanges by displaying arrows between the owners involved. This lets you quickly see when certain processes and threads interacted with each other.

To turn this on, click the **Toggle IPC Lines** dropdown () in the upper left toolbar, then choose **All** to see IPC arrows for all owners or **Select Owners** to see arrows only within the selection.



Pulses are indicated by orange arrows. Message sends are shown in blue while replies are shown in either green, for a successful reply, or red, for an error reply. In this example, the user has selected a Reply event from `hw_server Thread 5`. The arrow is green to indicate that the server replied to the client (`low_prio_client Thread 1`) after its request succeeded. Earlier in the timeline (i.e., to the left of the selected event), we can see a blue arrow between the same two owners but pointing in the opposite direction; this indicates the client's message send containing the request.

Timeline: Event labels

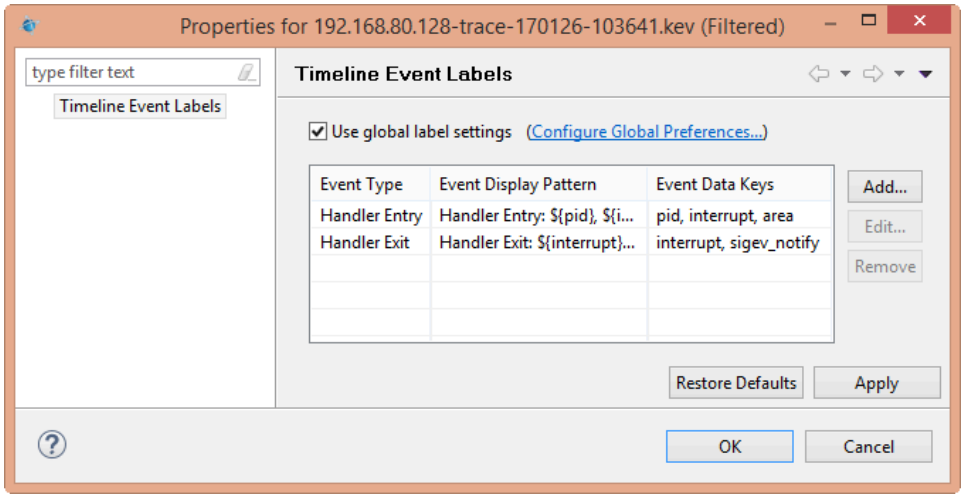
You can annotate timelines with event labels to easily distinguish between event types and see data associated with individual events. This latter feature is particularly helpful for matching Function Entry and Function Exit events with function names.

By default, no labels are drawn. To see them, click the **Toggle Labels** dropdown () and select from among these options:

- Priority Labels — for Running events, shows the owner's priority level
- State Labels — for State events, shows the owner's new state; this means you don't have to remember the colors for specific states
- State Icons — shows icons representing state changes; this saves space compared to names, but you must recognize the icons
- IPC Labels — for Communication events, shows the name of the thread receiving the message or pulse or being replied to
- Event Labels — for message-related events, shows entries into key functions (e.g., *read()*, *write()*)

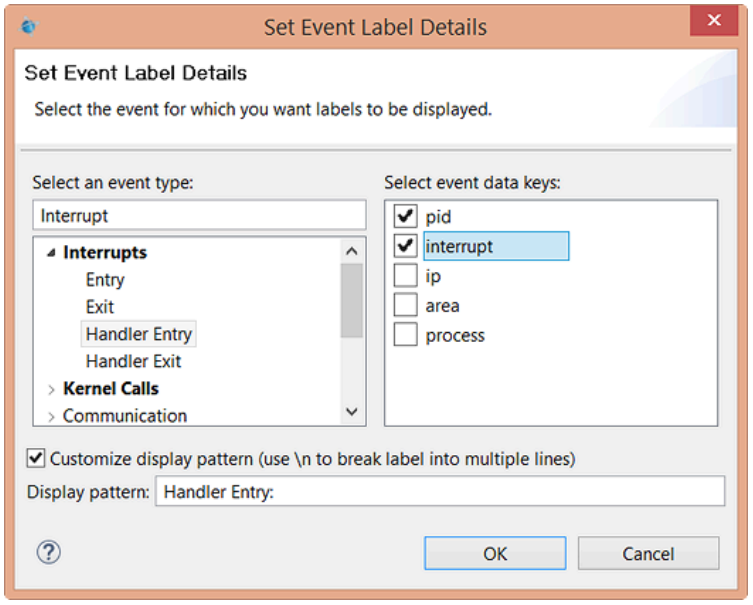
NOTE:
Event labels for all threads within a process are shown on the process's timeline. This can make it quite crowded, so it's often best to expand a process that you want to analyze, and view the labels on the thread timelines.

The last dropdown option, **Configure Event Labels**, opens a properties window that you lets you select the data fields to display in the event labels as well as how the data are formatted. Note that the settings in this window apply to the current log file only.



By default, **Use global label settings** is checked, so the System Profiler inherits the label display settings preconfigured for common events such as MsgSendV Enter. The link next to this box lets you change the default display settings used by all traces. The window opened by this link has the same table listing of individual settings and the same controls for configuring them as the first window. Note that you can also access the global label settings through **Window** > (and then)**Preferences** > (and then)**QNX** > (and then)**System Profiler** > (and then)**Timeline Event Labels**.

In both the trace-specific and global settings windows, the label display settings are shown in a table, with each row showing the data display pattern and data keys (fields) for a specific event type. On the right, there are buttons for adding, editing, and removing settings. Clicking **Add** opens the **Set Event Label Details** window.



This other window lets you select an event type, by expanding an event class on the left and checking the box for an individual type, then one or many data keys to display in the corresponding labels, by checking the relevant boxes on the right. It also lets you customize the display pattern. The default pattern consists of the event type followed by a comma-separated list of data keys. If you check **Customize display pattern**, the text field underneath becomes active and you can enter patterns containing literal text as well as these symbols:

- Data keys are specified by using `$data_key_name$`; in the Timeline pane, they're replaced with the actual event values for the given keys.
- To make the label text span multiple lines, use `\n`.

Timeline: Renaming processes and threads

Renaming processes and threads makes the Timeline pane more readable. Often, threads are unnamed, so this feature allows you to better identify them based on their purpose. If you run `tracelogger` in ring mode, the thread names are lost, so this way, you can assign your own names when viewing the trace results.

To rename an event owner, first select it (by left-clicking) in the Timeline display. Then, either right-click and choose `Rename` or press **F2**, and type the new name.

Page updated: August 11, 2025

Analyzing kernel activity with the System Profiler

QNX Tool Suite | Integrated Development Environment User's Guide | Developer | Setup

The System Profiler tool lets you view kernel event data gathered from a target. This tool displays graphs and tables that reveal kernel activity from all processes, down to the level of system calls and interrupts, over a fixed time period.

An application might run well, be very responsive, and be free of serious errors when executed in isolation but exhibit issues when introduced into a production environment. You must ensure that it performs and behaves well when running on your embedded system, which most likely contains many different applications and services. Kernel event tracing captures key system events that reveal how processes and threads interact with each other and can highlight performance bottlenecks and bad situations such as deadlock and improper signal handling.

The IDE automates starting kernel event traces and receiving, storing, and viewing the captured data. Through the System Profiler editor panes, you can extract specific data for different analysis use cases.

Page updated: August 11, 2025

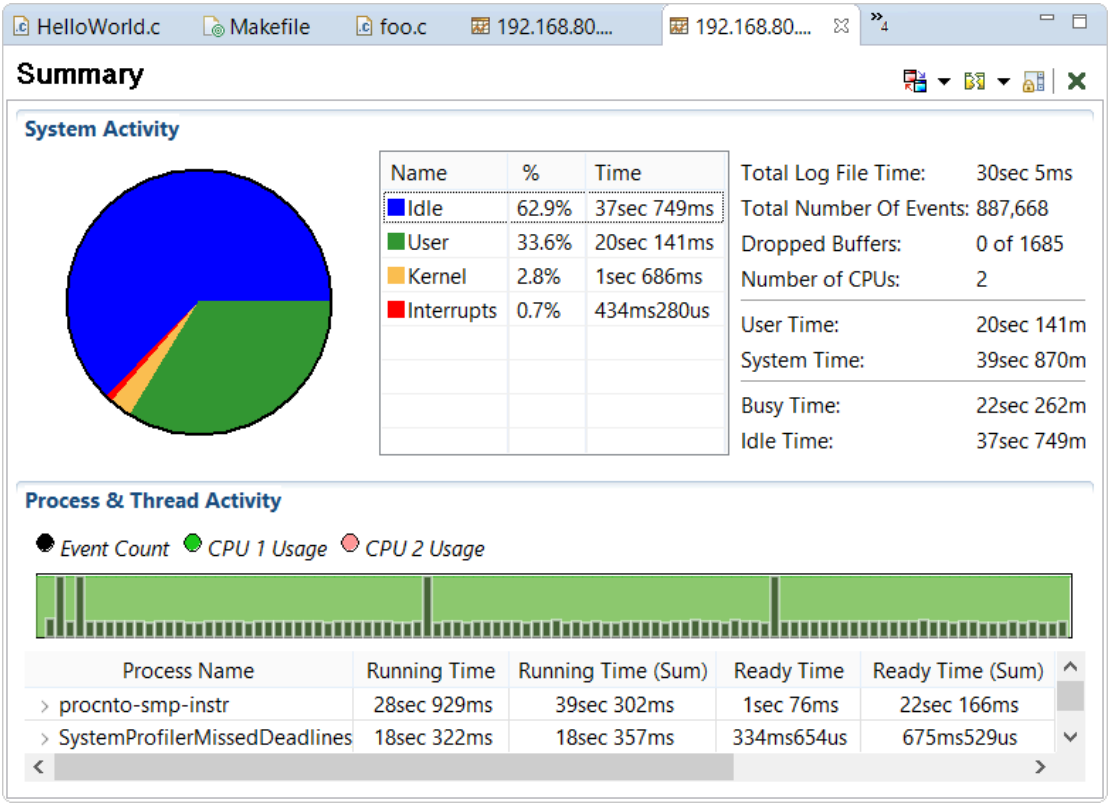
Interpreting trace data in the System Profiler

When a kernel event trace launched through the IDE finishes, you can open the new log file to see the file's data in the System Profiler editor. This editor is highly interactive and lets you see subsets of event data, which helps you understand component interaction and troubleshoot problems.

To manually open a trace file, choose **File** > (and then)**Open File** and navigate to and select the file. All kernel event logs generated for a specific target from the IDE are stored in ***workspace_dir*/*target_conn_name***.

NOTE:
If you run a kernel event trace outside of the IDE, you can still view the results with the System Profiler. You just need to copy the log file to a suitable host location and open it in the IDE.

By default, the Summary pane is shown. This pane reports the system-wide CPU time breakdown and general statistics about the kernel event trace, in the System Activity area. It also shows execution metrics for individual processes and threads, in the Process & Thread Activity area.



The editor lets you visualize the trace data in different ways, by selecting a particular pane through the Switch Pane dropdown in the upper right corner (▾). The panes that are most useful depend on which information you're trying to extract and which problems you're trying to solve. In the sections that follow, we describe how to locate specific events and filter the data in the System Profiler editor to perform common cases of analysis and troubleshooting.

Page updated: August 11, 2025

System Profiler

QNX Tool Suite | Integrated Development Environment User's Guide | Developer | Setup

The QNX System Profiler perspective displays kernel event trace data. Through the editor, you can view different aspects of the kernel activity and examine a time range of events in detail. There are also several views that list helpful information such as the order of events, thread state event statistics, and runtimes for client and server threads.

To see data, you must first run a kernel event trace, which you can do from the IDE or command line. The System Profiler reads kernel event log (**.kev**) files generated during traces. These files are automatically uploaded to the workspace when a trace finishes. After applying a filter or selecting a timeframe of events, you can save this subset of kernel event data to a new trace file, to keep only the most relevant information.

The sections that follow describe how to use the editor and views to find certain events and understand what the trace is telling you.

Page updated: August 11, 2025

System Profiler: user event formatting XML reference

This reference document defines attributes of the datakey specifier. For a detailed guide that walks through several examples, go to "*Formatting user events with the System Profiler*."

eventdefinition

The high-level tag for the XML document.

To see the legacy format, go to "eventclass (deprecated)."

datakey

The datakey tags are subtags of the `eventdefinitions` tag, and events are subtags of a datakey; this allows keys and enums to be assigned to multiple events. The *offset* attribute allows you to specify where the datakey resides within the event data. There is also a *wide_offset* attribute for handling non-user events, when you want to differentiate between simple and wide data.

The datakey specifies a format that is applied to trace events specified by the required event subtags. All other subtags are optional, but are useful for controlling how the specified format is applied.

Attributes:

One or more required.

Attribute	Description
format (string, required)	Define the elements and labels. Same as *sformat* but arrays aren't supported. For more information about `sformat`, go to "eventclass (deprecated)."
offset (number, optional)	Specify the region of data that the datakey should begin parsing the format at. Default value is 0.
wide_offset (number, optional)	Specify the region of data that the datakey should begin parsing the format at for wide data. Not recommended. Default value is 0.
show_masked_value (true/false, optional)	Displays hex values as well as enum strings. Only set this field to `false` if you haven't specified condition attributes for the field. Default value is `true`.

event (subtag)

One or more required.

Attributes:

Attribute	Name	Class ID	Description
class (number, required)			Required class ID of events this datakey applies to.
	CONTROL	1	The *_NTO_TRACE_CONTROL* class includes events related to the control of tracing itself.
	KER_CALL	2	The *_NTO_TRACE_KERCALL* is a pseudo-class that comprises all these classes.
	INT	3	The *_NTO_TRACE_INT* class includes events related to interrupts.
	PR_TH	4	The *_NTO_TRACE_THREAD* class includes events related to state changes for threads.
	SYSTEM	5	The *_NTO_TRACE_SYSTEM* class includes events related to the system as a whole.

Attribute	Name	Class ID	Description
	USER	6	The *_NTO_TRACE_USER* class includes custom events that your program creates.
	COMM	7	The *_NTO_TRACE_COMM* class includes events related to communication.
	QUIP	8	The *_NTO_TRACE_QUIP* class is the QNX Unified Instrumentation Platform class.
	SEC	9	The *_NTO_TRACE_SEC* class includes events related to security.
	QVM	10	The hypervisor event class includes events omitted by the hypervisor. To learn how to support these events, go to the information on updating hypervisor event trace descriptions in the hypervisor documentation.
id (number, required)			Required event ID that this datakey applies to.

Example:

The event tag in the example datakey targets the user class events 65 and 15.

```
<eventdefinitions>
  <datakey format="%2u1x ipicmd %2u1x ipicmd2" offset="0"
wide_offset="0" show_masked_value="true">
    <event class="6" id="65" />
    <event class="6" id="15" />
  </datakey>
</eventdefinitions>
```

bit mask (subtag)

You can perform bitmask ANDing to extract a subset of the bits from the field with a matching `name` attribute; this affects how this value is matched to enums. Bitmasks are intended to help isolate bitfields packed together into other data sizes. For example, you might pack together a 6-bit and 10-bit value into a 16-bit integer. A bitmask would help you split these values up into their own fields. If the mask ends with zeroes, like `0xff00`, then the 0's on the bitmasked value are truncated. For example, if you had the field `ipicmd (0x7cff)`, and bitmasked it with `0xff00`, the resulting value would be `0x7c` and not `0x7c00`. This truncation allows you to more easily use enums to match against bitfields, as if you were accessing the bitfield in code.

Attributes:

Attribute	Description
name (string, required)	The label of the element to bitmask. Although typically required, this attribute is not necessary if the sformat attribute of the parent datakey only has one field (ie: `<datakey sformat="%4u1x onlyfield" ... >`).
key (string, deprecated - use `name` instead)	Serves the same function as the `name` field which is now preferred. Can still be used for compatibility with older file formats.
value (number, required)	Value to bitwise AND with the specified field.

Example:

In this example, the `ipicmd` field is masked with `0xff`.

```
<eventdefinitions>
  <datakey format="%2u1x ipicmd %2u1x ipicmd2" offset="0"
wide_offset="0" show_masked_value="true">
    <event class="6" id="65" />
    <bitmask name="ipicmd" value="0xff" />
  </datakey>
</eventdefinitions>
```

condition (subtag)

Display condition that controls whether the format specified in the parent datakey is applied to the event data, based on the mask, value, or both.

NOTE:
You should at minimum specify a mask or a value. You can also specify both a mask and a value.

CAUTION:
Conditions that match against fields specified by the format being conditionally applied are ignored. In the following example, the condition matching is ignored because the *ipicmd* field is specified in the format of the datakey with the condition. To work around this, only use condition subtags targeting fields defined in other datakeys.

```
<eventdefinitions>
  <datakey format="%1u1x ipicmd" offset="0" wide_offset="0"
show_masked_value="false">
    <event class="6" id="65" />
    <condition key="ipicmd" mask="!0x0001" />
  </datakey>
</eventdefinitions>
```

Attributes:

Attribute	Description
name (string, required)	The label of the field that this condition matches against. Use field names contributed by other datakeys. Optional if only one label in format string. Otherwise, label of element to be applied to.
key (string, deprecated - use `name` instead)	Serves the same function as the `name` field which is now preferred. Can still be used for compatibility with older file formats.
value (number, optional)	Value to compare against. The condition evaluates to true if the element (or masked element, if mask is present) equals the value.  **NOTE:** The value for this attribute can be prepended with ! to invert the match.
mask (number, optional)	Number to bitwise AND with the specified element. If no value attribute is present, the condition will evaluate to true if the masked element equals the mask itself.

Example:

In the following example, The value in `ipicmd` would be bitwise AND with `0x0001`, and depending on whether the resulting value is `0x0001`, the contributed value would be formatted as `dstring1` or `dstring2`.

```
<eventdefinitions>
  <datakey format="%2u1x ipicmd" offset="0" wide_offset="0"
show_masked_value="true">
    <event class="6" id="65" />
  </datakey>
  <datakey format="%1s0 dstring1" offset="0" wide_offset="0"
show_masked_value="true">
    <event class="6" id="65" />
    <condition key="ipicmd" mask="0x0001" />
  </datakey>
  <datakey format="%1s0 dstring2" offset="0" wide_offset="0"
show_masked_value="true">
    <event class="6" id="65" />
    <condition name="ipicmd" mask="!0x0001" />
  </datakey>
</eventdefinitions>
```

enum (subtag)

Specify strings to display.

> **NOTE:**
> The enum matching is performed after the bitmask tag is applied to the target value.

Attributes:

Attributes support either mask or value, but not both.

Attribute	Description
name (string, required)	Optional if only one label in format string. Otherwise, label of element to be applied to.
value (number, optional)	Value to compare against. Can be prepended with ! to invert the match.
mask (number, optional)	Mask value to compare against. If you provide a mask, the following operation is used for matching: `raw_value & mask == mask`. Can be prepended with ! to invert the match.
string(string, required)	String to display if mask or value match.
default(string, optional)	String to display if no other enum matches.

Example:

```
<eventdefinitions>
  <datakey format="%2u1x ipicmd %2u1x ipicmd2" offset="0"
wide_offset="0" show_masked_value="true">
    <event class="6" id="65" />
    <enum name="ipicmd" mask="0x100" string="1_bit_8" />
    <enum name="ipicmd" mask="0x200" string="1_bit_9" />
    <enum name="ipicmd" mask="0x400" string="1_bit_10" />
    <enum name="ipicmd" default="1_default" />
    <enum name="ipicmd2" value="0x1" string="2_val_1" />
    <enum name="ipicmd2" value="0x2" string="2_val_2" />
    <enum name="ipicmd2" value="0x4" string="2_val_3" />
    <enum name="ipicmd" default="2_default" />
  </datakey>
</eventdefinitions>
```

eventclass (deprecated)

The `eventclass` tag provides a simple way to format events, but has been deprecated in favor of the `datakey` tag which is more powerful. For more information, go to “eventdefinition.”

event (subtag)

One or more required.

Attribute	Description	Format
sformat	Define the elements and labels. In general, an event is described as a serial series of event payload definitions. The label becomes the field name shown in the IDE. Where: • **size** – Size in bytes. Valid values are: 1, 2, 4, 8. • **signed** – Specifies whether the value is signed (s) or unsigned (u). • **count** – The number of items to read (ie., an array). If the <size> is 1 and there is no formatting, then the <count> can be 0; this accommodates NULL-terminated strings. • **format** – (optional) A hint as to how to format this value: d(decimal), x(hexadecimal), o(octal), c(character). • **label** – String label that can't contain the % character.	`%<size><signed><count><format><label>`
id	Required event ID of events this datakey applies to.	NA

Example:

The `%2u1x ipicmd` indicates that the field will be labeled `ipicmd`. It's an unsigned, single value; formatted in hexadecimal; and made up of 2 bytes of data.

```
<eventdefinitions>
  <eventclass id="6" name="User Events">
    <event id="65" sformat="%2u1x ipicmd %2u1x ipicmd2" />
    <event id="15" sformat="%2u1x ipicmd %2u1x ipicmd2" />
  </eventclass>
</eventdefinitions>
```

Page updated: August 11, 2025

System Resources

You can track the resource usage of target processes through the **System Resources** view. Specifically, this view lets you compare the CPU, memory, or file descriptor usage of different processes.

You can display one set of statistics at a time, by clicking the dropdown button (▽) in the upper right corner of the view, then selecting one of three menu options:

System Uptime

This display shows the start time, CPU usage time, and CPU usage as a percentage of total uptime, for each process. These statistics help you monitor process performance.

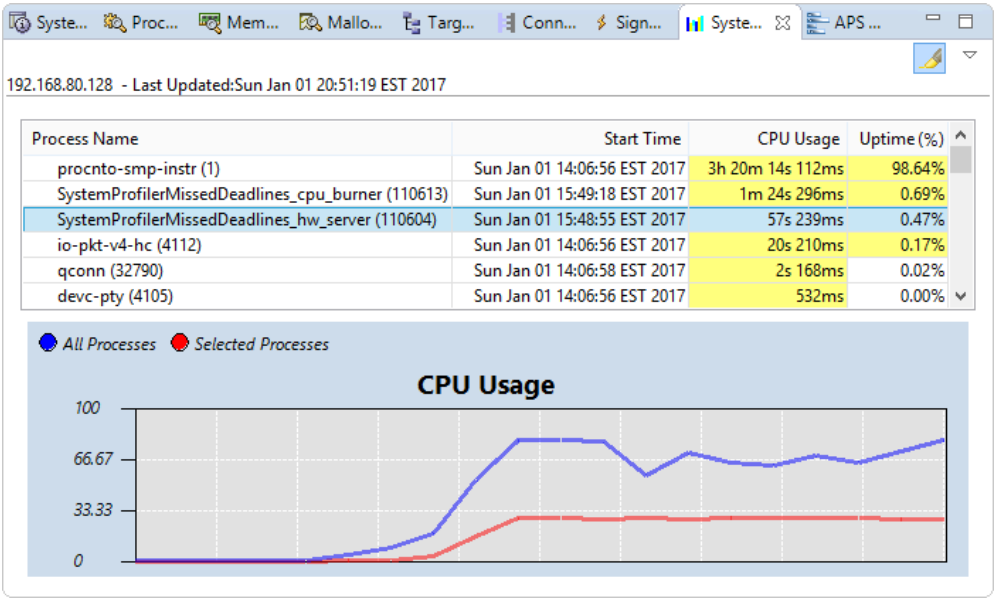
General Resources

These statistics include the relative CPU usage, data segment size, and number of open file descriptors for each process, which help you monitor resource usage.

Memory Resources

These statistics include the sizes of the heap and stack segments and of the code and data segments for the program and for the loaded libraries. This memory information lets you monitor memory consumption of processes.

You can sort the process list by any displayed metric, by clicking the corresponding column header. To easily spot changed values, click the highlight button (✎) in the upper right toolbar of the view. The view then colors in the cells containing any values that changed since the last update. You can change the highlight color in the System Information preferences, by selecting **Window** > (and then)**Preferences** > (and then)**General** > (and then)**Appearance** > (and then)**Colors and Fonts** > (and then)**System Information** > (and then)**Highlight color**. The default color is yellow, as seen in this sample screenshot that shows the System Uptime statistics, with the process list sorted by CPU Usage:



The **CPU Usage** graph is always shown, regardless of which statistics set is displayed. This line graph shows historical CPU usage by plotting the percentage of cycles consumed by various processes over time. By default, both the **All Processes** and **Selected Processes** lines are enabled, but no processes in the statistics table are selected. So initially, the chart shows only the total CPU consumption of all processes. You can click one or more table rows to see the combined CPU usage for multiple processes. You can also deselect one of the buttons just above the chart, to hide the corresponding graph line.

Page updated: August 11, 2025

System Summary

The **System Summary** view provides information about the overall state of a target. This information includes specifications of the target machine, system-wide memory usage, and a listing of active processes that includes performance statistics.

Individual panes within the view present these different areas of information, as described below. This view is handy for monitoring memory consumption at the system level and comparing changes in CPU usage for different processes.

System Specifications

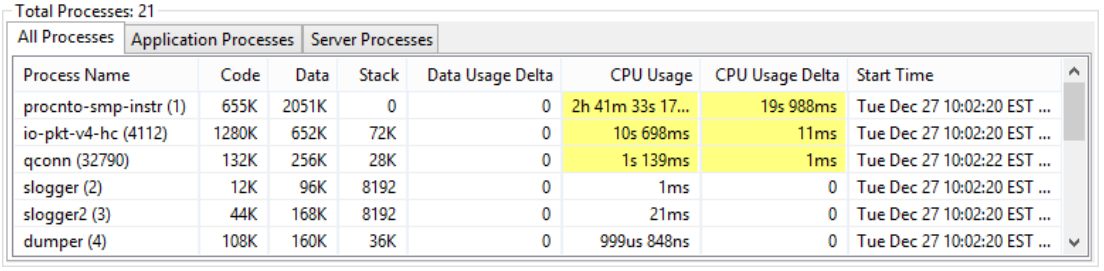
This top pane lists some of the target's hardware and software details, including the hostname, board architecture type, OS version, boot date and time, and CPU types and speeds. The CPU details are shown for each processor core.

System Memory

This pane shows the used, free, and total system memory, in numerical and graphical form.

Total Processes

In this pane, the number of active processes is displayed next to the title, while realtime statistics for process memory and CPU usage are shown in the table area below. Each table row lists a process's name and its code, data, and stack segment sizes, data segment delta, total CPU usage since starting, CPU usage delta, and start date and time. The different tabs let you see tables that list all processes, just the application processes, or just the server processes.



Process Name	Code	Data	Stack	Data Usage Delta	CPU Usage	CPU Usage Delta	Start Time
procnto-smp-instr (1)	655K	2051K	0	0	2h 41m 33s 17...	19s 988ms	Tue Dec 27 10:02:20 EST ...
io-pkt-v4-hc (4112)	1280K	652K	72K	0	10s 698ms	11ms	Tue Dec 27 10:02:20 EST ...
qconn (32790)	132K	256K	28K	0	1s 139ms	1ms	Tue Dec 27 10:02:22 EST ...
slogger (2)	12K	96K	8192	0	1ms	0	Tue Dec 27 10:02:20 EST ...
slogger2 (3)	44K	168K	8192	0	21ms	0	Tue Dec 27 10:02:20 EST ...
dumper (4)	108K	160K	36K	0	999us 848ns	0	Tue Dec 27 10:02:20 EST ...

You can sort the process list by any displayed metric, by clicking the corresponding column header. To easily spot changed values, click the highlight button ( ) in the upper right toolbar of the view. The view then colors in the cells containing any values that changed since the last update. You can change the highlight color in the System Information preferences, by selecting **Window** > (and then)**Preferences** > (and then)**General** > (and then)**Appearance** > (and then)**Colors and Fonts** > (and then)**System Information** > (and then)**Highlight color**.

Page updated: August 11, 2025

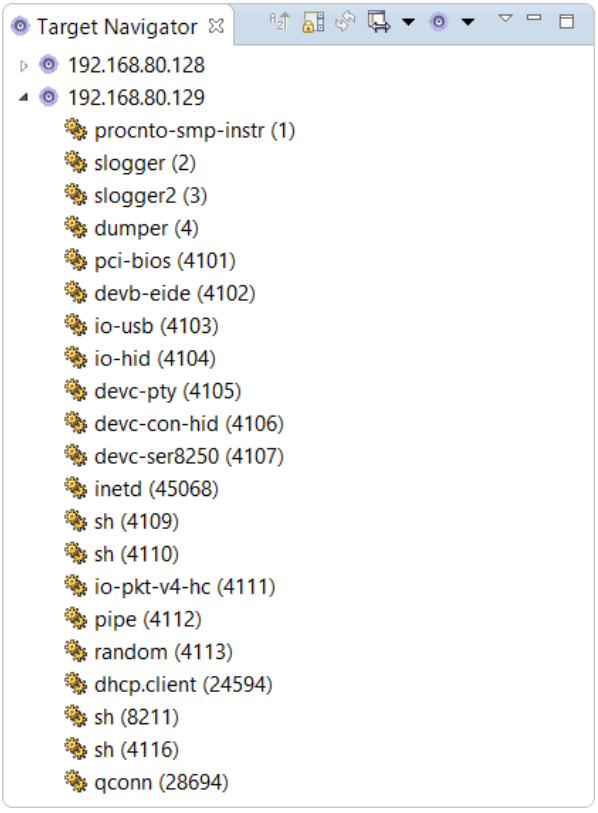
Target Navigator

The **Target Navigator** lists the targets and, for the connected ones, the processes running on them. Through this view's controls, you can configure kernel event tracing and adaptive partitioning for targets and send signals to processes, attach the debugger to them, and adjust their scheduling properties.

This view is also used to select the process for which you want to view runtime data. When you click a process entry in the **Target Navigator**, several other System Information views are updated to show that process's statistics, as explained in "How data are displayed."

The QNX System Information and QNX System Profiler perspectives automatically open this view but you can access it at any time through: **Window** > (and then)**Show View** > (and then)**Other** > (and then)**QNX Targets** > (and then)**Target Navigator**

The targets and processes are listed in a tree:



By clicking the button for the customization dropdown ( ) in the toolbar along the top, you can access options for:

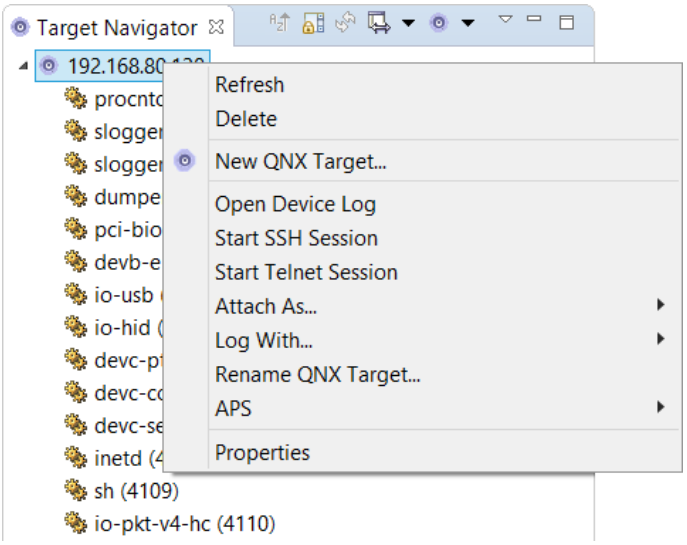
- grouping processes by PID family
- sorting processes by PID or name
- changing the refresh rate

The toolbar also has buttons for:

- reversing the sorting order ( )
- disabling or re-enabling auto-refresh ( )
- manually refreshing the display ( )
- starting or configuring kernel event tracing ( )
- creating a new target connection ( )
- minimizing and maximizing the view

Interacting with targets

You can interact with targets by right-clicking their entries and using the context menu. This menu allows you to refresh a target's process list, rename or delete its connection, or create a new connection. It also has options to open the device log (which appears in the **Console** view), launch an SSH or Telnet session, attach the debugger to a process, start kernel event tracing, and manage adaptive partitions settings:

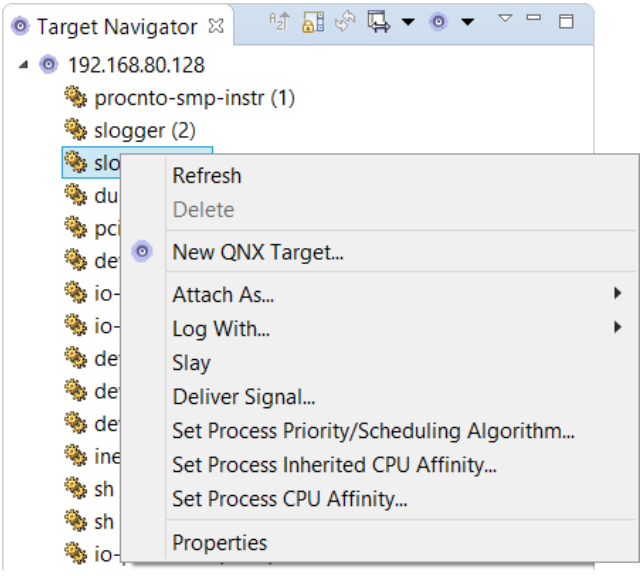


If you point to **Attach As**, the resulting submenu has two options: **C/C++ QNX Attach to Process** and **Attach Configurations**. For the first option, the IDE opens a window for selecting a process from a list. When you then do this, the IDE creates a launch configuration called `attach.`*`process_name`* and tries to launch it by attaching the debugger to the running process. For the second option, it opens a window for managing launch configurations, in which you can create your own attach configuration.

The **Log With** option lets you run a kernel event trace. From the **APS** submenu, you can configure the adaptive partitioning scheduler, as described below in "Configuring APS". Finally, the **Properties** option opens the **QNX Qconn Target** panel for configuring the target connection.

Interacting with processes

When you right-click a process entry, you see different context menu options than for targets:



Some target-level operations are still accessible from the process-level menu. These include refreshing the process list, creating a new connection, launching a kernel event trace, and displaying the target properties. Here, the **Attach As** option doesn't display a window with a process list—the IDE just creates the new attach configuration and tries to launch it to start debugging the selected process.

The middle portion of the menu shows options for interacting with processes. For example, you can send signals by clicking **Deliver Signal** and then, in the resulting popup window, selecting the appropriate signal from the dropdown list and clicking **OK**. You can explicitly terminate a process by selecting **Slay**. There are also options to adjust a process's priority and its inherited and non-inherited processor affinities.

Each of these process options opens a popup window for setting the corresponding property. For information about scheduling priorities and policies, see the "Thread scheduling" section in the *System Architecture* guide. For information about the processor affinity settings that associate threads with a particular core, see the "Processor affinity, clusters, runmasks, and inherit masks" section in the *Programmer's Guide*.

Configuring APS

NOTE:

QNX OS 8.0 does not support adaptive partitioning; if your target is based on 8.0, then APS can't be configured and these controls will be ineffective. They do work if your target is based on an earlier OS verison.

The APS submenu in the context (right-click) menu for a target contains these options:

Set APS Security

This option opens a popup window that provides checkboxes for enabling APS security flags. These flags determine who can add partitions and modify their attributes, and are described in the “Security” section for the *SchedCtl()* entry in the *C Library Reference* for the OS version that your target is based on.

Note that after you set a flag, you can't unset it through this window, so the flag doesn't appear anymore. To unset it, you must reboot the target.

Set APS Parameters

The popup window opened by choosing this option lets you modify the length of the averaging window, which is the time interval used to calculate CPU usage, and enable specific bankruptcy flags, which determine how the system reacts when a partition exhausts its critical budget and hence, becomes bankrupt; these flags are described in the “Handling bankruptcy” section for the *SchedCtl()* entry for the OS version that your target is based on.

Note that after you set a flag, you can't unset it through this window, so the flag doesn't appear anymore. To unset it, you must reboot the target.

Modify Existing Partition

This option lets you select a partition and set its budget and critical budget. The first setting is a percentage of CPU usage while the second is in milliseconds.

Create New Partition

Choosing this option opens a window into which you can enter a name, parent partition, budget, and critical budget to create a new partition. The new partition's budget is taken from its parent's budget.

Page updated: August 11, 2025

Supported target types

This IDE release supports several target types. You can work with physical or virtual targets connected to the host by an IP or a serial link, or virtual machines (VMs) generated by the IDE.

When you access the Launch Target dropdown and click **New Launch Target**, the resulting dialog shows the possible target types. The options with "QNX" in the title are meant for QNX OS developers, and include the following:

QNX Serial Port Connection

If you've configured a target for serial communication, you can create a **QNX Serial Port Connection** target to debug programs over a serial link. The resulting target connection can't be used with any analysis tool, though, such as Memory Analysis or System Profiler.

QNX Target

If you've configured a target for IP communication, the **QNX Target** option lets you create a target connection in which the IDE talks to the `qconn` agent at the specified target port. This target type works with physical boards or pre-existing VMs, and allows you to run and debug your programs or analyze them using any of the integrated tools.

Information on creating a QNX target is given in "Creating a target connection", and information on configuring it is given in "Configuring a QNX target".

QNX Virtual Machine Target

This target type lets you create a VM that runs QNX OS. All tasks related to running, debugging, and analyzing programs with any tool are supported. You can generate QNX VMs for many kinds of commercial third-party VM platforms.

Creating your own VM allows you to develop QNX OS applications before target boards are available, and to build VMs that have the latest QNX software packages. Instructions on doing so are given in "Creating a QNX virtual machine".

NOTE:

The **New Launch Target** dialog shows other options that are inherited from the Workbench desktop environment. These aren't meant to be used in the QNX Momentics IDE and will likely be removed in the future. For more information on the other options, refer to the Eclipse IDE documentation.

Page updated: August 11, 2025

Unit Testing

The IDE lets you use commercial test frameworks to write unit tests and then execute them by launching a project. While running a test program, you can use the Code Coverage tool to determine how much of the code is exercised (covered) by your tests.

The IDE supports these test frameworks:

- Boost.Test Library
- GoogleTest Framework
- Qt Testing Framework

This release of QNX SDP includes the GoogleTest Framework and the IDE is configured to work with it. You can therefore write test programs based on the Google framework without any extra setup. With the other two test frameworks, the IDE can parse the results of their test programs but you must unpackage and compile these frameworks on your host and manually configure the IDE to use them.

When you run a test program based on any of these frameworks, the IDE visually presents the test results. The integrated Code Coverage tool lets you measure the quality of your tests by reporting which areas of code they exercise. Thus, you can find and fix bugs in your code and limitations in your test programs in a single run-edit-compile cycle.

Page updated: August 11, 2025

Updating the IDE

The IDE includes an auto-update feature that lets you check for and install updates to the IDE itself.

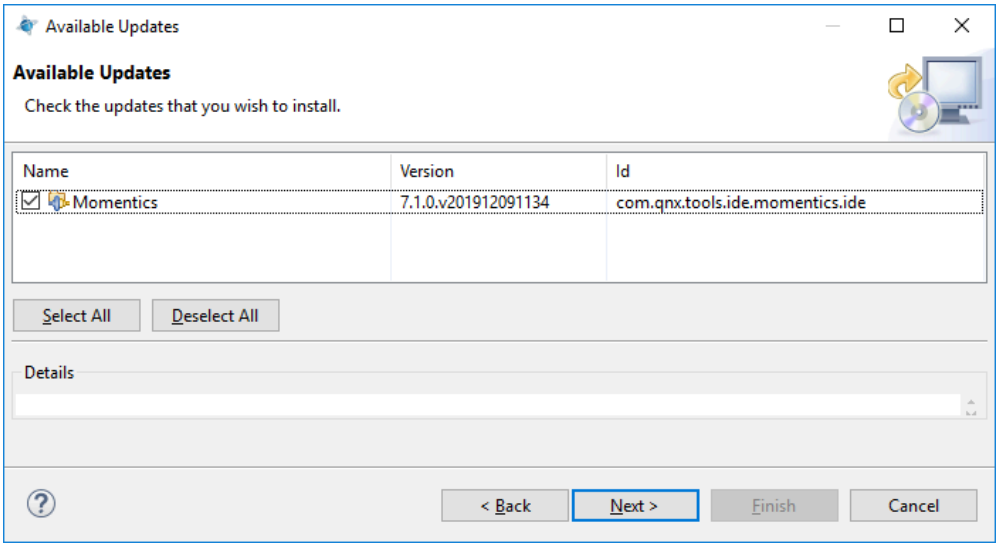
You must have an active Internet connection for auto-update to work, because it must talk to a server at **qnx.com** to check for the available updates and download the ones you choose to install.

To check for and install updates:

1. Select **Help** > (and then)**Check for Updates**.

   The IDE displays a message, `Contacting Software Sites...`, with a progress bar in the lower right corner while it contacts the server and asks it about available updates. Typically, this only last a few seconds and the update wizard then opens.

   The first dialog shown in the wizard, **Available Updates**, lists the updates found:



   NOTE:
   If no updates are available, you won't see the wizard but instead a popup window advising you that no updates are available and providing a link to the **Available Update Sites** window in the QNX preferences, which lets you configure other sites from which to read content.

2. In the list area, check the boxes for the updates you want to install.
   Each row in the list provides the name, version (which is the build ID), and package ID of the update. Sometimes, a summary of the selected update is provided in the **Details** panel shown below the list area. There's also a link, entitled `More`, that appears when an update is selected, and when clicked, opens a window that lists various properties for it.

3. Click **Next** to advance to the **Update Details** dialog.
   This dialog displays the contents of the updates you previously selected, in a tree-like display that lets you expand each plugin in the update package and see any other plugins within it. Below this display area is a **Size** field that lists the size of the selected plugin (if it's known), and a **Details** panel with a `More` link that serve the same purpose as in the first dialog.

4. Click **Next** to advance to the **Review Licenses** dialog.
   This dialog displays information from the End User License Agreements (EULAs) that apply to the update package. In the left box, each entry lists the first line from a EULA, and you can expand the entry to see the plugins to which it applies. Clicking a particular EULA displays the first line of its text in the right box. You must click the **I accept** radio button below these boxes to continue.

5. Click **Finish**.
   The IDE displays a message, `Updating Software...`, with a progress bar in the lower right corner while it downloads and tries to install the selected updates.

If the auto-update operation succeeds, a popup window appears confirming the operation's success and stating that the IDE must be restarted for the update to take effect. You can click **Restart Now** to do so right away, or **No** to defer the restart. A restart is necessary for the IDE to reload the plugins.

Upgradeable workspace

You can import existing workspaces from QNX SDP 7.1,7.0,6.6, or 6.5 SP1 into Momentics. All of your project's custom settings are preserved.

Page updated: August 11, 2025

Valgrind

QNX Tool Suite | Integrated Development Environment User's Guide | Developer | Setup

Valgrind (pronounced "val-grinned") is a third-party tool suite that supports runtime analysis of programs. The IDE provides controls for configuring some of these tools, allowing you to track a program's dynamic memory operations, measure its heap usage, and more.

Here, we describe how to use the **Tools** tab in the launch configuration properties to configure the Valgrind tools that are integrated with the IDE. The integrated tools are:

- Massif—Measures heap usage over time and where memory is being allocated
- Memcheck—Detects memory management problems by checking all reads and writes of memory
- Helgrind—Finds thread synchronization problems in programs that use pthreads
- Cachegrind—Measures a program's reads and writes and its cache misses

QNX SDP actually includes the binaries for all tools in the Valgrind tool suite release. You can analyze your programs with any of them from the command line. For information on doing so and examples for the four tools mentioned above, see the `valgrind` entry in the *Utilities Reference* and the online Valgrind User Manual.

How to configure Valgrind tools

When you click the **Valgrind** radio button in the **Tools** tab, the IDE displays the **Tool to run** dropdown, in which you can select which of the supported Valgrind tools to configure. The UI controls shown below the dropdown consist of five tabs:

1. **General Options**—displays some of the more widely used options that apply for all tools.

   The **Basic Options** panel has the following fields:

   Trace children on exec
   : Whether to trace into any process images created by *exec*()*. Valgrind automatically traces into any process copies created by *fork()*, but you must enable the **Trace children on exec** option to make it follow execution into new process images created by *exec*()*.

   Child silent after fork
   : Whether to remain silent and *not* show debugging or logging output for child processes created by *fork()*. Enabling this option makes the output less confusing for multiprocess programs and is especially useful when **Trace children on exec** is enabled.

   Run __libc_freeres on exit
   : Whether to run the *__libc_freeres* routine provided by the C library (**libc.so**) when a process that uses the library exits. This routine releases all of the memory allocated by the library for its own uses, and was developed to prevent some leak checkers such as Valgrind from falsely reporting leaks in **libc** when a leak check is done at a process exit.

     If your program runs fine with Valgrind but encounters a segmentation fault while exiting, you can disable this option to prevent the running of *__libc_freeres* and avoid the crashing (although there may be falsely reported space leaks in **libc.so**).

   The following fields are shown in the **Error Options** panel:

   Demangle C++ names
   : Whether to automatically demangle C++ names. When enabled, Valgrind attempts to translate encoded C++ names into a format closer to their original names.

     If you're writing suppression files manually, you should disable this option to see the mangled names in your error messages. This form must be used in suppression files.

   Callers in stack trace
   : Sets the maximum number of entries to show in stack traces for program locations. This value doesn't affect the overall number of errors reported, just the call chain depth.

     Higher settings make Valgrind run slower and use more memory, but are necessary for programs with deep call chains. The default value is 12, and the maximum value is 500.

   Limit errors reported
   : Whether to stop reporting errors after 10,000,000 in total or 1,000 different types have occurred. Enabling this option limits the performance impact on programs with many errors.

   Show errors below main

Whether to show any functions below *main()* in stack traces. Typically, such functions do C library-related work and other operations that are uninteresting for program analysis.

Max stack frame size (B)

The maximum size, in bytes, of a stack frame. If the stack pointer moves by more than this amount, Valgrind assumes that the program is switching to a different stack.

You should only use this option if Valgrind's debugging output directs you to do so. In that case, it will tell you the new threshold you should specify.

Set main thread's stack size (B)

The stack size, in bytes, for the program's initial thread. This setting doesn't affect the size of thread stacks, as Valgrind doesn't allocate those.

You may need to use this option and **Max stack frame size** together. You have to work out the suitable main thread stack size yourself (e.g., if your program encounters a segmentation fault)—Valgrind won't tell you like it will for the thread stack frame size.

Note that the **Run dsymutil (Mac OS X)** checkbox has been disabled because QNX OS is not supported for macOS targets.

There's also the **Extra Options** text field, which lets you manually specify any extra general options you want to pass to the `valgrind` utility.

2. **Supressions**—lets you name up to 100 suppression files that list any errors you want suppressed in the output. On QNX OS targets, the default suppression file is **/usr/lib/valgrind/default.supp**. This file contains comments explaining the syntax required to specify an error to suppress.

   There are buttons for opening file navigators and selecting suppression files from the workspace or filesystem, and for removing a listed file.

3. ***Tool* Options**—Here, *Tool* is the name of the tool selected in the **Tool to run** dropdown.

   For the **Massif** selection, the tab provides these options:

Option	Description
Profile heap	Whether to profile heap blocks.
Administrative bytes per block	The number of administrative bytes to use per heap block. This option is effective only if **Profile heap** is enabled.
Profile stack	Whether to profile stacks. Enabling this option greatly slows down Massif but produces stack usage data that's easy to read.
Profile memory at page level	Whether to profile memory at the page level rather than at the level of blocks allocated by *malloc()*.
Allocation tree depth	The depth of the allocation tree, which lists the exact parts of code responsible for allocating heap memory.
Heap allocation threshold	The relative size threshold for heap blocks to be reported individually in the results. Allocations for blocks smaller than this threshold are aggregated into a single results entry.
Allocation peak inaccuracy	The percentage by which memory allocation must exceed the previous peak to be considered the new peak. Smaller values mean greater accuracy in the peak checking, but for values near 0%, Massif runs very slowly.
Time units	The time unit for measuring execution progress, one of: ◦ `i` for machine instructions ◦ `ms` for milliseconds since the program's start ◦ `B` for the number of heap bytes currently allocated
Detailed snaphost frequency	How often the heap snapshots are *detailed*; a value of ***N*** means Massif makes every ***N***th snapshot a detailed one. These snapshots include stack traces of block allocation points, and a breakdown of heap memory by allocation point.
Max snapshots	The maximum number of snapshots (***N***) to keep in the results. Whenever Massif takes another snapshot and exceeds this limit, half of them are deleted. For longer programs, the final number of snapshots is between ***N***/2 and ***N***.

Option	Description
Minimum heap block alignment	The minimum alignment, in bytes, required for all allocated heap blocks. When a program asks for ***N*** bytes, Massif rounds ***N*** up to the nearest multiple of the value specified by this option.
Allocation functions	List of the functions to be treated like heap allocation functions. This is useful for functions that are wrappers to *malloc()* or *new*. There's a button for opening a dialog box to specify a new function to add to the list, and a button for removing the selected function from it.

For the **Memcheck** selection, the tab provides these options:

Option	Description
Check for memory leaks	Whether to check for memory leaks. When this option is enabled, Valgrind checks for leaks when the program exits and prints a leak summary.
Leak resolution	The degree to which stack traces must match to be reported as the same leak, one of: ◦ `low`—only the first two entries, which represent the bottom two functions in the call chain, must match ◦ `med`—the first four entries must match ◦ `high`—all entries must match (this is the default setting) This option is effective only if **Check for memory leaks** is enabled.
Freelist size (blocks)	The size, in bytes, of the space used to store recently freed blocks. A larger freelist means a longer time period in which Memcheck can store freed blocks and detect invalid accesses to them, but also an increased memory footprint.
Show reachable blocks	Whether to include *reachable* and *indirect* memory leaks in the results. Reachable leaks are blocks for which a pointer to the start of the memory can be found but the memory was never freed. Indirect leaks occur when the blocks that point to the memory are lost themselves.
Allow partial loads	Whether to allow 32-, 64-, 128- and 256-bit naturally aligned loads from addresses for which some bytes are addressable and others are not. When this option is enabled, such loads don't produce an error. When it's disabled, loads from partially invalid addresses produce an illegal-address error, and the resulting bytes are marked as initialized.
Undefined value errors	Whether to report undefined value use errors. Disabling this option speeds up Memcheck.
Track origins of uninitialized values	Whether to track the origin of uninitialized memory (e.g., a heap block allocation, stack variable, or client request) when it's used dangerously. Enabling this option severely degrades performance, but greatly reduces the effort needed to find the cause of undefined value use errors. If you enable this option, you must also enable **Undefined value errors**.
GCC 2.96 workarounds	Whether to assume that reads and writes some small distance below the stack pointer are due to bugs in GCC 2.96 and hence, not to report them. You should enable this option only if you're using a version of QNX SDP older than 6.5.0 SP1. Otherwise, don't use it because real errors can be overlooked.
Minimum heap block alignment	The minimum alignment, in bytes, required for all allocated heap blocks. When a program asks for ***N*** bytes, Massif rounds ***N*** up to the nearest multiple of the value specified by this option.
Show possibly lost blocks in leak check	Whether to include *possible* memory leaks in the results. These are blocks for which a pointer to the middle but not the start of the memory can be found.

Option	Description
Fill malloc'd areas with given value (0x)	The byte pattern with which to fill blocks allocated by *malloc()*, *new*, etc, but not by *calloc()*. This option is useful when trying to solve obscure memory corruption problems.
Fill free'd areas with given value (0x)	The byte pattern with which to fill blocks released by *free()*, *delete*, etc. This option is useful when trying to solve obscure memory corruption problems.
Ignore Ranges	List of address ranges to be ignored by Memcheck's addressability checking. There's a button for opening a dialog box to specify a new address range to add to the list, and a button for removing a selected range from it.

For the **Helgrind** selection, the tab provides these options:

Option	Description
Track lockorders	Whether to perform lock order consistency checking. When this option is enabled, Helgrind reports any inconsistencies in the order in which threads acquire locks. Inconsistent locking can lead to lock cycles, which create potential deadlocks that can lead to hard-to-diagnose failures.
History level	The historical data to keep about conflicting memory accesses, one of: ◦ `full`—two stack traces are reported for data races: the full trace of the current access to a memory location, and the partial trace (up to 8 entries) of the previous access ◦ `approx`—Helgrind reports the full trace of the current access, and two earlier traces such that the previous access happened between them ◦ `none`—no data is kept about previous accesses
Conflict cache size	Size of the cache to keep for storing data about conflicting memory accesses. This value is the number of memory addresses for which access history is kept, not the number of bytes. This option is effective only when **History Level** is `full`. The minimum value is 10,000, and the maximum value is 30,000,000 (thirty million).

For the **Cachegrind** selection, the tab provides these options:

Option	Description
Profile Cache Accesses/Misses	Whether to collect cache access and miss counts. You can't disable both this option and **Profile Branch Instructions/Mispredictions** (because no information would be collected).
Profile Branch Instructions/Mispredictions	Whether to collect branch instruction and misprediction counts. You can't disable both this option and **Profile Cache Accesses/Misses** (because no information would be collected).
Manually Set Cache Specifications	These checkboxes enable spinners for manually setting the total size, associativity, and line size of the following caches: ◦ **I1 Cache**—first-level instruction cache ◦ **D1 Cache**—first-level data cache ◦ **L2 Cache**—second-level unified cache The total size and line size values are in bytes while the associativity value must be a power of two between 1 (for direct mapping) and 1024 (for 1024-way associativity).

4. **Symbols**—provides controls for manually specifying the locations of debug symbols for any shared libraries used by your programs. Normally, the `valgrind` utility should be able to find these symbols by communicating with the symbol server, which runs on the host.

   You need to use this tab only if:

- The debug symbols are on the host and Valgrind isn't producing any results and is terminating with a message saying that it's unable to connect to the server.

  In this case, you need to ensure that the server is running at the correct port and IP address, as explained in the "Valgrind" subsection of the "Configuring shared library support" topic.

- The debug symbols are on the target, or you tried configuring the symbol server but still don't see any results (and hence, you must upload the symbols to the target).

  In this case, you can specify the target directories where the symbols are stored, as described in the "Manually loading debug symbols on the target" subsection of that same topic.

5. **Debugging Options**—displays options that are useful for debugging.

   Track open file descriptors on exit

   Whether Valgrind will print a list of open file descriptors on exit or on request. Along with each file descriptor, Valgrind will print also a stack backtrace of where the file was opened and file descriptor details such as the filename or socket details.

   Show time stamp

   Whether each message is preceded with the elapsed time since startup, expressed as days, hours, minutes, seconds, and milliseconds.

   Trace system calls

   Trace signals

   Detection of self-modifying code

   Control Valgrind's detection of self-modifying code. You can select one of these options:

   - `none`—no detection
   - `stack`—detect self-modifying code on the stack (which is used for nested functions)
   - `all`—detect self-modifying code everywhere
   - `all-non-file`—detect self-modifying code everywhere except in file-backed mappings

   For information about the effect on a program's performance and behavior for each of the settings, see the `smc-check` description in the Valgrind User Manual.

   Verbosity level

   The verbosity level, which determines the extra information reported on various aspects of the program. You can choose `none`, `low`, or `high`.

Page updated: August 11, 2025

What's new in the IDE?

QNX Tool Suite | Integrated Development Environment User's Guide | Developer | Setup

The QNX Momentics IDE version 8 is build on top of Eclipse 2023-06 and supports the features of QNX OS 8.0.

The major changes are as follows:

- Support for QNX SDP 8.0
- Built on Eclipse Platform 4.28 a and CDT 11.2 (simrel 2023-06)
- Compliant to the C11, C17, C++17 and C++20 standards (IDE editor)
- 3rd party software has been updated to fix any known security issues

NOTE:

- The IDE is backwards compatible. It supports command line tools and runtimes from previous versions, including QNX SDP 7.0 and 7.1.
- This release also includes several fixes for bugs that were present in the IDE 7.1 release. For the list of bugs fixed, see the *QNX Momentics IDE 8.0 Release Notes*.

Page updated: August 11, 2025

Writing code

When you create a non-empty project, the IDE creates the directory structure and generates default source files and makefiles based on the project type that you select. For a QNX Executable project, the source file containing the *main()* function is automatically opened for editing.

NOTE:
If you haven't switched to the C/C++ perspective, we recommend doing so for writing code. This perspective gives you access to the Eclipse-based C/C++ Development Toolkit (CDT), which lets you efficiently manage and develop projects based on the C and/or C++ programming language.

The default C/C++ editor contains syntax highlighting and code assist features to help you write C and C++ code; information on working with this editor can be found in the **Task** > (and then)**Writing code** entry of the *C/C++ Development User Guide*. The IDE actually contains many other built-in editors; you can select a particular editor for opening a file, as explained in "Editors".

To display, browse, and open existing project resources—header files, source code files, and makefiles, you can expand the project's entry in the **Project Explorer** view and double-click individual file entries. Using this same view or the **File** menu, you can add any of these resource types as needed.

Page updated: August 11, 2025

Writing and building test programs

The test frameworks supported by the IDE let you write programs in C++ that test C or C++ code. Here, we explain the QNX project setup required to write a test program using the GoogleTest (GTest) Framework. For the other frameworks, you should read their online documentation to learn the coding and building steps needed to write test programs.

NOTE:
The code being tested can belong to an application or a library.

To write and build a test program based on the GTest framework:

1. In the project that contains the code that you want to run unit tests on, create the additional source files and write the code necessary for the test cases.
   The online GoogleTest documentation explains the GTest framework concepts and how to write a test program, starting from individual assertions and working towards a complete program with many test cases. Here, we list a few of the key steps for writing test programs:
   - Store the test program files in a specific project folder (e.g., **test**), to keep them separate from the main program, for ease of maintenance.
   - In the header file for your test program, you must include the GTest header file, as follows:
     ```
     #include <gtest/gtest.h>
     ```
   - In your test program's source files, you must include any header files declaring the functions that you want to test. This is because the test macros must be able to see the prototypes of the functions that they're testing.
   - In the GTest framework, individual tests are grouped into test cases. We recommend putting each test case in its own source file, to keep together all of the tests that exercise a specific program area. If you're using test fixtures to share data between tests, each source file will define an individual class, in which you can implement routines that prepare the data before the tests defined in the same file get run, and that release any resources used by the tests after they finish running.
   - You must call *RUN_ALL_TESTS()* exactly once in the program (in the *main()* function), even if your test cases are defined in multiple files.
2. Modify your makefile to dynamically link the GTest library.

   Although the installer from this latest SDP release includes the GTest library (**libgtest.so**), you must still add the library to your project. You must also link your program against the **libregex** library, by including the `-Bdynamic -l regex` flags in your linking command. Details on these linker flags are given in the `q++`, `qcc` entry in the *Utilities Reference*.

   If you're building your programs with an older SDP version, we don't recommend trying to use the GTest library shipped with this latest SDP release to develop unit test programs. This would be complicated because you might have to modify the GTest library to compile properly and/or acquire other libraries needed by GTest but not shipped with the platform version that you're building with.
3. Click the Build button ().

The IDE attempts to build the test program and displays the build output in the **Console** window at the bottom. If the build succeeds, you'll see the binary files listed in the project area in the **Project Explorer**. If the build fails, the **Problems** window shows any errors (in red) and warnings (in yellow).

Page updated: August 11, 2025