

# 分析中央处理器使用率

Qt Creator与Linux Perf工具集成，可用于分析嵌入式设备上应用程序的CPU和内存使用情况，并且在有限的程度上，在Linux桌面平台上。性能分析器使用与Linux内核捆绑在一起的Perf工具定期拍摄应用程序的调用链快照，并在时间轴视图中或火焰图中可视化它们。

## 使用性能分析器


性能分析器通常需要能够找到所涉及的二进制文件的调试符号。

配置文件生成生成具有单独调试符号的优化二进制文件，通常应用于分析。

若要手动设置生成配置以提供单独的调试符号，请编辑项目生成设置：

1. 若要还为在发布模式下编译的应用程序生成调试符号，请选择“**项目**”，然后在“**单独的调试信息**”字段中选择“**启用**”。
2. 选择“**是**”重新编译项目。

可以通过以下方式启动性能分析器：

- 选择“**分析>性能分析器**”以分析当前应用程序。
- 选择“ (开始)”按钮以从**性能分析器**启动应用程序。




**注意：** 如果数据收集未自动启动，请选择“ (收集配置文件数据)”按钮。

开始分析应用程序时，将启动该应用程序，并且性能分析器将立即开始收集数据。这由“**记录**”字段中运行的时间指示。但是，由于数据通过Perf工具和与Qt Creator捆绑在一起的额外帮助程序传递，并且都动态缓冲和处理它，因此数据可能会在生成后几秒钟到达Qt Creator。此延迟的估计值在“**处理延迟**”字段中给出。

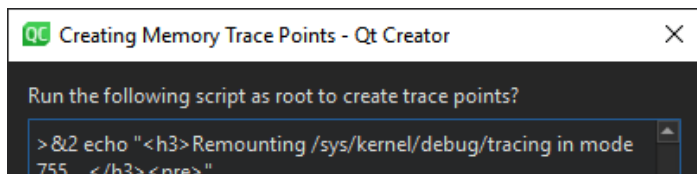
在您选择“**停止收集配置文件数据**”按钮或终止应用程序之前，将收集数据。

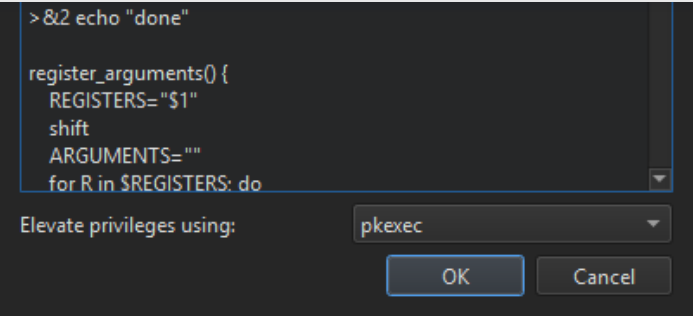
选择“**停止收集配置文件数据**”按钮以禁用启动应用程序时自动启动数据收集。配置文件数据仍将生成，但Qt Creator将丢弃它，直到您再次选择该按钮。

## 分析设备上的内存使用情况

若要创建跟踪点以分析目标设备上的内存使用情况，请选择“**分析>性能分析器选项**”>“**创建内存跟踪点**”，或在“**性能分析器**”工具栏上选择“”。

在“**创建内存跟踪点**”对话框中，可以修改要运行的脚本。





如果需要 root 权限才能以 root 身份运行脚本，请在“提升权限使用”字段中选择要使用的权限。

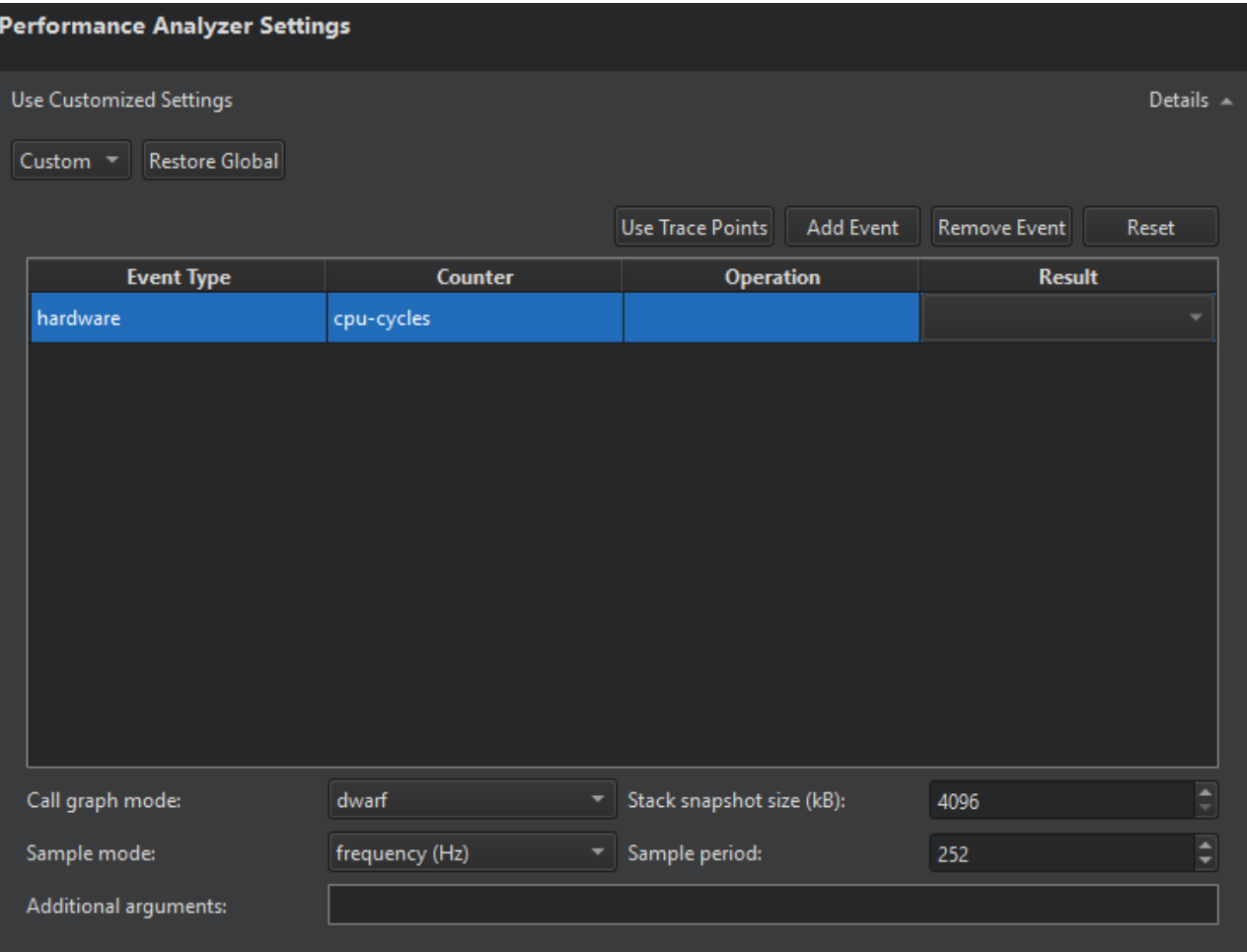
选择“确定”以运行脚本。

若要为跟踪点添加事件，请参阅[选择事件类型](#)。

您可以记录内存跟踪，以便在时间线的示例行中查看使用情况图，并在火焰图中查看内存分配、峰值和释放。

## 指定性能分析器设置

若要为性能分析器指定全局设置，请选择“[编辑](#)”>“[首选项](#)”>“[分析器](#)”>“[CPU 使用情况](#)”。对于每个运行配置，还可以使用专用设置。选择“[项目](#)”>“[运行](#)”，然后选择“[性能分析器设置](#)”旁边的“[详细信息](#)”。



To edit the settings for the current run configuration, you can also select the dropdown menu next to the **Collect profile data** button.

### Choosing Event Types

In the **Events** table, you can specify which events should trigger the Performance Analyzer to take a sample. The most common way of analyzing CPU usage involves periodic sampling, driven by hardware performance counters that react to the number of instructions or CPU cycles executed. Alternatively, a software counter that uses the CPU clock can be chosen.

More specialized sampling, for example by cache misses or cache hits, is possible. However, support for it depends on specific features of the CPU involved. For those specialized events, you can give more detailed sampling instructions in the **Operation** and **Result** columns. For example, you can choose a **cache** event for **L1-dcache** on the **load** operation with a result of **misses**. That would sample L1-dcache misses on reading.

Select **Remove Event** to remove the selected event from the table.

Select **Use Trace Points** to replace the current selection of events with trace points defined on the target device and set the **Sample mode** to **event count** and the **Sample period** to . If the trace points on the target were defined using the **Create Trace Points** option, the Performance Analyzer will automatically use them to profile memory usage.<sup>1</sup>

Select **Reset** to revert the selection of events, as well as the **Sample mode** and **Sample period** to the default values.

## Choosing a Sampling Mode and Period

In the **Sample mode** and **Sample period** fields, you can specify how samples are triggered:

- Sampling by **event count** instructs the kernel to take a sample every times one of the chosen events has occurred, where is specified in the **Sample period** field.nn
- Sampling by **frequency (Hz)** instructs the kernel to try and take a sample times per second, by automatically adjusting the sampling period. Specify in the **Sample period** field.nn

High frequencies or low event counts result in more accurate data, at the expense of a higher overhead and a larger volume of data being generated. The actual sampling period is determined by the Linux kernel on the target device, which takes the period set for Perf merely as advice. There may be a significant difference between the sampling period you request and the actual result.

In general, if you configure the Performance Analyzer to collect more data than it can transmit over the connection between the target and the host device, the application may get blocked while Perf is trying to send the data, and the processing delay may grow excessively. You should then change the **Sample period** or the **Stack snapshot size**.

## Selecting Call Graph Mode

In the **Call graph mode** field, you can specify how the Performance Analyzer recovers call chains from your application:

- The **Frame Pointer**, or , mode relies on frame pointers being available in the profiled application and will instruct the kernel on the target device to walk the chain of frame pointers in order to retrieve a call chain for each sample.f.p
- The **Dwarf** mode works also without frame pointers, but generates significantly more data. It takes a snapshot of the current application stack each time a sample is triggered and transmits that snapshot to the host computer for analysis.
- The **Last Branch Record** mode does not use a memory buffer. It automatically decodes the last 16 taken branches every time execution stops. It is supported only on recent Intel CPUs.

Qt and most system libraries are compiled without frame pointers by default, so the frame pointer mode is only useful with customized systems.

## Setting Stack Snapshot Size

The Performance Analyzer will analyze and *unwind* the stack snapshots generated by Perf in dwarf mode. Set the size of the stack snapshots in the **Stack snapshot size** field. Large stack snapshots result in a larger volume of data to be transferred and processed. Small stack snapshots may fail to capture call chains of highly recursive applications or other intense stack usage.

## Adding Command Line Options For Perf

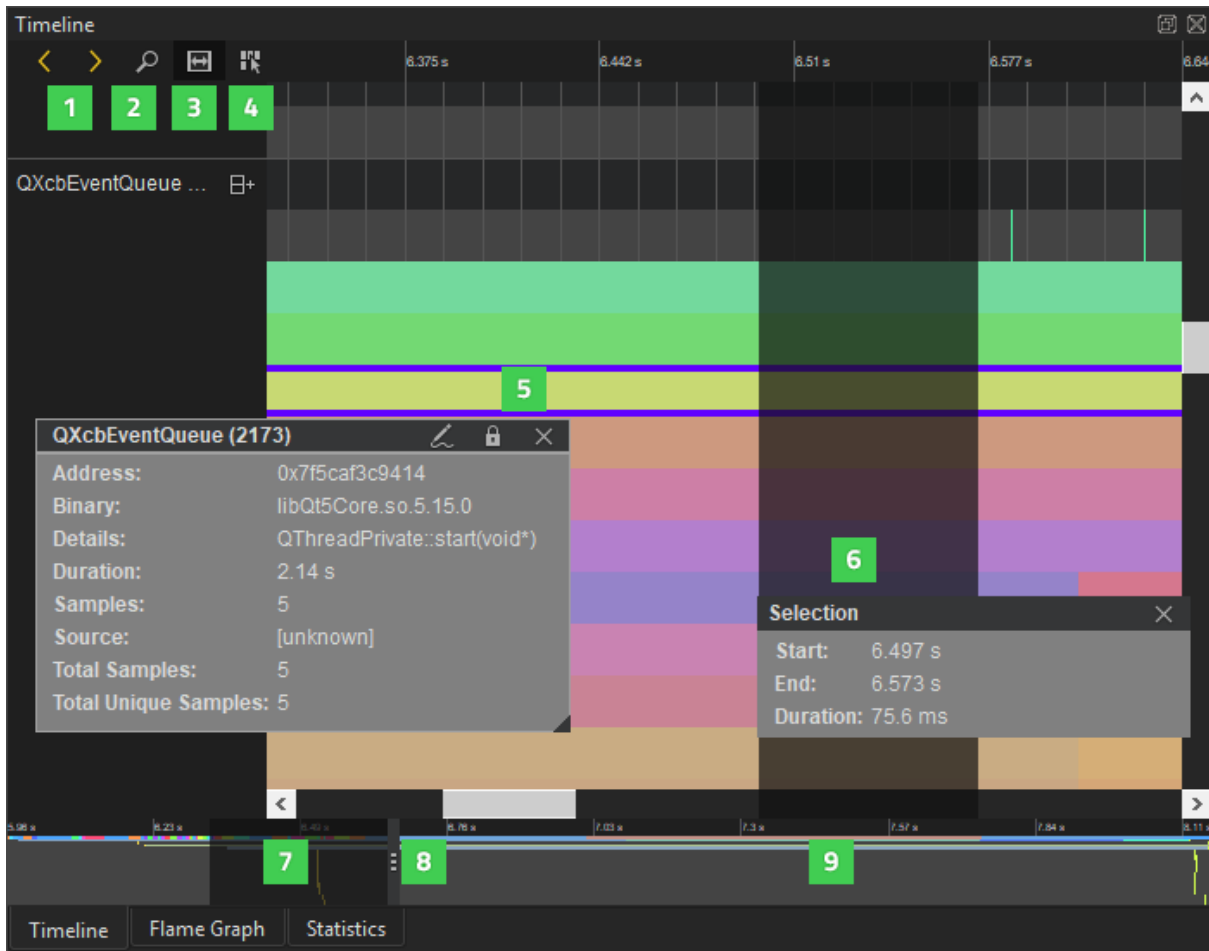
You can specify additional command line options to be passed to Perf when recording data in the **Additional arguments** field. You may want to specify or to reduce the processing delay. However, those options are not supported by all versions of Perf and Perf may not start if an unsupported option is given. `--no-delay --no-buffering`

## Resolving Names for JIT-compiled JavaScript Functions

Since version 5.6.0, Qt can generate perf.map files with information about JavaScript functions. The Performance Analyzer will read them and show the function names in the **Timeline**, **Statistics**, and **Flame Graph** views. This only works if the process being profiled is running on the host computer, not on the target device. To switch on the generation of perf.map files, add the environment variable to the **Run Environment** and set its value to `.QV4_PROFILE_WRITE_PERF_MAP1`

## Analyzing Collected Data

The **Timeline** view displays a graphical representation of CPU usage per thread and a condensed view of all recorded events.



Each category in the timeline describes a thread in the application. Move the cursor on an event (5) on a row to see how long it takes and which function in the source it represents. To display the information only when an event is selected, disable the **View Event Information on Mouseover** button (4).

The outline (9) summarizes the period for which data was collected. Drag the zoom range (7) or click the outline to move on the outline. You can also move between events by selecting the **Jump to Previous Event** and **Jump to Next Event** buttons (1).

Select the **Show Zoom Slider** button (2) to open a slider that you can use to set the zoom level. You can also drag the zoom handles (8). To reset the default zoom level, right-click the timeline to open the context menu, and select **Reset Zoom**.

## Selecting Event Ranges

You can select an event range (6) to view the time it represents or to zoom into a specific region of the trace. Select the **Select Range** button (3) to activate the selection tool. Then click in the timeline to specify the beginning of the event range. Drag the selection handle to define the end of the range.

You can use event ranges also to measure delays between two subsequent events. Place a range between the end of the first event and the beginning of the second event. The **Duration** field displays the delay between the events in milliseconds.

To zoom into an event range, double-click it.

To remove an event range, close the **Selection** dialog.

## Understanding the Data

Generally, events in the timeline view indicate how long a function call took. Move the mouse over them to see details. The details always include the address of the function, the approximate duration of the call, the ELF file the function resides in, the number of samples collected with this function call active, the total number of times this function was encountered in the thread, and the number of samples this function was encountered in at least once.

As the Perf tool only provides periodic samples, the Performance Analyzer cannot determine the exact time when a function was called or when it returned. You can, however, see exactly when a sample was taken in the second row of each thread. The Performance Analyzer assumes that if the same function is present at the same place in the call chain in multiple consecutive samples, then this represents a single call to the respective function. This is, of course, a simplification. Also, there may be other functions being called between the samples taken, which do not show up in the profile data. However, statistically, the data is likely to show the functions that spend the most CPU time most prominently.

If a function without debug information is encountered, further unwinding of the stack may fail. Unwinding will also fail for some symbols implemented in assembly language. If unwinding fails, only a part of the call chain is displayed, and the surrounding functions may seem to be interrupted. This does not necessarily mean they were actually interrupted during the execution of the application, but only that they could not be found in the stacks where the unwinding failed.

JavaScript functions from the QML engine running in the JIT mode can be unwound. However, their names will only be displayed when is set. Compiled JavaScript generated by the Qt Quick Compiler can also be unwound. In this case the C++ names generated by the compiler are shown for JavaScript functions, rather than their JavaScript names. When running in interpreted mode, stack frames involving QML can also be unwound, showing the interpreter itself, rather than the interpreted JavaScript.QV4\_PROFILE\_WRITE\_PERF\_MAP

Kernel functions included in call chains are shown on the third row of each thread.

The coloring of the events represents the actual sample rate for the specific thread they belong to, across their duration. The Linux kernel will only take a sample of a thread if the thread is active. At the same time, the kernel tries to honor the requested event period. Thus, differences in the sampling frequency between different threads indicate that the thread with more samples taken is more likely to be the overall bottleneck, and the thread with less samples taken has likely spent time waiting for external events such as I/O or a mutex.

## Viewing Statistics

Statistics

Address	Function	Source Location	Binary Location	Occurrences	Recursion in Percent	Samples	Samples in Percent	Self Samples	Self in Percent
0x00007f5caf3...	QThreadPri...		libQt5Core.so...	6	0	6	9.2	0	0
0x00007f5caf4...	QHashData...		libQt5Core.so...	1	0	1	1.5	0	0
0x00007f5caf5...	QLoggingR...		libQt5Core.so...	1	0	1	1.5	0	0
0x00007f5caf5...	QFileSyste...		libQt5Core.so...	1	0	1	1.5	0	0
0x00007f5caf5...	QFactoryLo...		libQt5Core.so...	3	0	3	4.6	0	0
0x00007f5caf5...	QFactoryLo...		libQt5Core.so...	1	0	1	1.5	0	0
0x00007f5caf5...	QFactoryLo...		libQt5Core.so...	1	0	1	1.5	0	0
0x00007f5caf5...	QLibraryPriv...		libQt5Core.so...	3	0	3	4.6	0	0
0x00007f5caf5...	QLibraryPriv...		libQt5Core.so...	3	0	3	4.6	0	0
0x00007f5caf5...	QLibraryPriv...		libQt5Core.so...	3	0	3	4.6	0	0
0x00007f5caf5...	QLibraryPriv...		libQt5Core.so...	1	0	1	1.5	0	0
0x00007f5caf5...	QLibraryPriv...		libQt5Core.so...	1	0	1	1.5	0	0
0x00007f5caf5...	QLibraryPriv...		libQt5Core.so...	3	0	3	4.6	0	0
0x00007f5caf5...	QEventLoop...		libQt5Core.so...	3	0	3	4.6	0	0
0x00007f5caf5...	QCoreApplication...		libQt5Core.so...	1	0	1	1.5	0	0
0x00007f5caf5...	QCoreApplication...		libQt5Core.so...	1	0	1	1.5	0	0
0x00007f5caf5...	QCoreApplication...		libQt5Core.so...	1	0	1	1.5	0	0
0x00007f5caf5...	QCoreApplication...		libQt5Core.so...	1	0	1	1.5	0	0
0x00007f5caf5...	QCoreApplication...		libQt5Core.so...	2	0	2	3	0	0

Address	Callee	Occurrences	Occurrences in Percent	Address	Caller	Occurrences	Occurrences in Percent
0x00007f5caa...	QDBusConne...	1	16.6	0x00007f5cae5c...	start_thread	6	100
0x00007f5cab...	QXcbEventQu...	3	50				
0x00007f5cab...	QXcbEventQu...	2	33.3				

Timeline Flame Graph Statistics

The **Statistics** view displays the number of samples each function in the timeline was contained in, in total and when on the top of the stack (called ). This allows you to examine which functions you need to optimize. A high number of occurrences might indicate that a function is triggered unnecessarily or takes very long to execute.self

Click on a row to move to the respective function in the source code in the code editor.

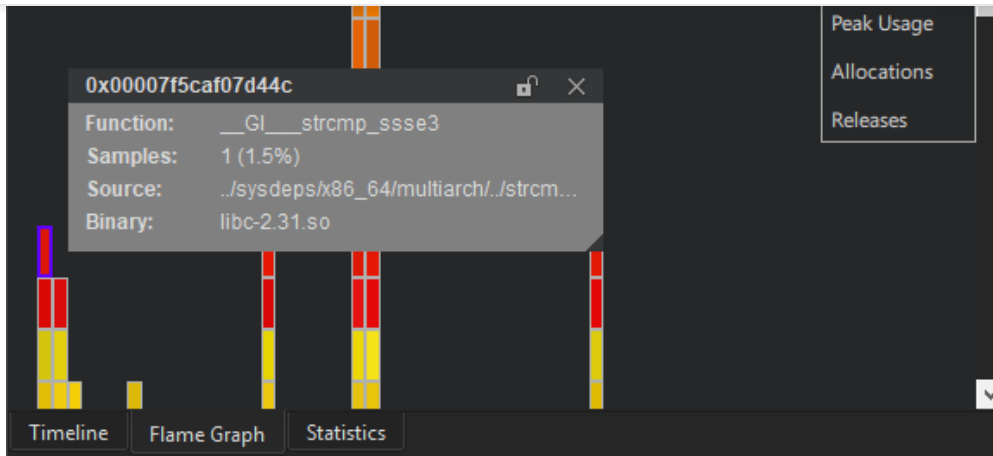
The **Callers** and **Callees** panes show dependencies between functions. They allow you to examine the internal functions of the application. The **Callers** pane summarizes the functions that called the function selected in the main view. The **Callees** pane summarizes the functions called from the function selected in the main view.

Click on a row to move to the respective function in the source code in the code editor and select it in the main view.

To copy the contents of one view or row to the clipboard, select **Copy Table** or **Copy Row** in the context menu.

## Visualizing Statistics as Flame Graphs





The **Flame Graph** view shows a more concise statistical overview of the execution. The horizontal bars show an aspect of the samples taken for a certain function, relative to the same aspect of all samples together. The nesting shows which functions were called by which other ones.

The **Visualize** button lets you choose what aspect to show in the **Flame Graph**.

- **Samples** is the default visualization. The size of the horizontal bars represents the number of samples recorded for the given function.
- In **Peak Usage** mode, the size of the horizontal bars represents the amount of memory allocated by the respective functions, at the point in time when the allocation's memory usage was at its peak.
- In **Allocations** mode, the size of the horizontal bars represents the number of memory allocations triggered by the respective functions.
- In **Releases** mode, the size of the horizontal bars represents the number of memory releases triggered by the respective functions.

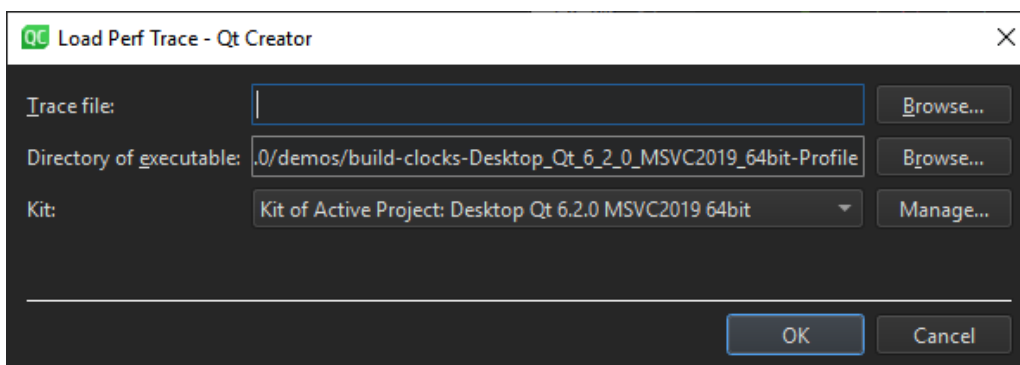
The **Peak Usage**, **Allocations**, and **Releases** modes will only show any data if samples from memory trace points have been recorded.

## Interaction between the views

When you select a stack frame in either of the **Timeline**, **Flame Graph**, or **Statistics** views, information about it is displayed in the other two views. To view a time range in the **Statistics** and **Flame Graph** views, select **Analyze > Performance Analyzer Options > Limit to the Range Selected in Timeline**. To show the full stack frame, select **Show Full Range**.

## Loading Perf Data Files

You can load any files generated by recent versions of the Linux Perf tool and view them in Qt Creator. Select **Analyze > Performance Analyzer Options > Load perf.data File** to load a file.perf.data



The Performance Analyzer needs to know the context in which the data was recorded to find the debug symbols. Therefore, you have to specify the kit that the application was built with and the folder where the application executable is located.

The Perf data files are generated by calling `perf record`. Make sure to generate call graphs when recording data by starting Perf with the option `--call-graph`. Also check that the necessary debug symbols are available to the Performance Analyzer, either at a standard location (or next to the binaries), or as part of the Qt package you are using. `perf record --call-graph /usr/lib/debug`

checking the output of or for the recorded Perf data files.`perf report``perf script`

## Loading and Saving Trace Files

You can save and load trace data in a format specific to the Performance Analyzer with the respective entries in **Analyze > Performance Analyzer Options**. This format is self-contained, and therefore loading it does not require you to specify the recording environment. You can transfer such trace files to a different computer without any tool chain or debug symbols and analyze them there.

## Troubleshooting

The Performance Analyzer might fail to record data for the following reasons:

- › Perf events may be globally disabled on your system. The preconfigured **Boot2Qt** images come with perf events enabled. For a custom configuration you need to make sure that the file contains a value smaller than . For maximum flexibility in recording traces you can set the value to . This allows any user to record any kind of trace, even using raw kernel trace points.`/proc/sys/kernel/perf_event_paranoid=1`

The way to enable Perf events depends on your Linux distribution. On some distributions, you can run the following command with root (or equivalent) privileges:

```
echo -e"kernel.perf_event_paranoid=-1\nkernel.kptr_restrict=0" | sudo tee /etc/sysctl.d/10-perf.conf
```

- › The connection between the target device and the host may not be fast enough to transfer the data produced by Perf. Try modifying the values of the **Stack snapshot size** or **Sample period** settings.
- › Perf may be buffering the data forever, never sending it. Add or to the **Additional arguments** field.`--no-delay --no-buffering`
- › Some versions of Perf will not start recording unless given a certain minimum sampling frequency. Try with a **Sample period** value of 1000.
- › On some devices, in particular various i.MX6 Boards, the hardware performance counters are dysfunctional and the Linux kernel may randomly fail to record data after some time. Perf can use different types of events to trigger samples. You can get a list of available event types by running on the device and then choose the respective event types in the settings. The choice of event type affects the performance and stability of the sampling. The event is a safe but relatively slow option as it does not use the hardware performance counters, but drives the sampling from software. After the sampling has failed, reboot the device. The kernel may have disabled important parts of the performance counters system.`perf listcpu-clocksoftware`
- › Perf might not be installed. The way to install it depends on your Linux distribution. For example, you might try the following commands:
  - › On Ubuntu 22.04: `sudo apt install linux-tools-$(uname -r)`
  - › On Debian: `apt install linux-perf`

Output from the helper program that processes the data is displayed in **General Messages**.

Some information is displayed in **Application Output** even if the Performance Analyzer displays error messages.

[◀ Detecting Memory Leaks with Heob](#)

[Analyzing Code with Cppcheck ▶](#)

© 2022 The Qt Company Ltd. Documentation contributions included herein are the copyrights of their respective owners. The documentation provided herein is licensed under the terms of the [GNU Free Documentation License version 1.3](#) as published by the Free Software Foundation. Qt and respective logos are trademarks of The Qt Company Ltd in Finland and/or other countries worldwide. All other trademarks are property of their respective owners.

Qt

DOCUMENTATION

Qt

The Qt Company

f

Contact Us

Company

About Us

Investors

Newsroom

Careers

Office Locations

Licensing

Terms & Conditions

Open Source

FAQ

Support

Support Services

Professional Services

Partners

Training

For Customers

Support Center

Downloads

Qt Login

Contact Us

Customer Success

Community

Contribute to Qt

Forum

Wiki

Downloads

Marketplace

© 2022 The Qt Company

Feedback

Sign In