

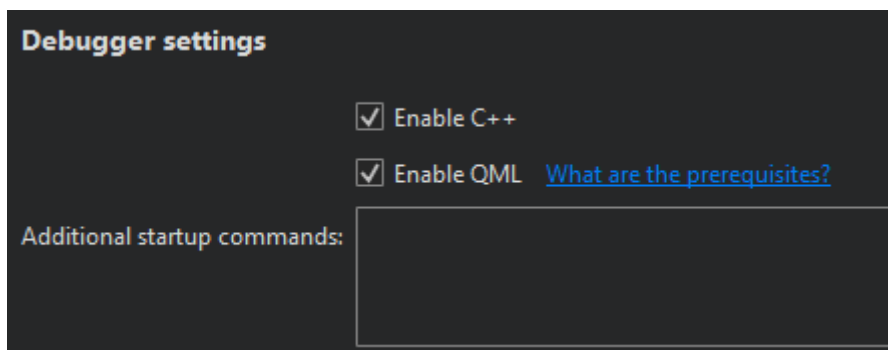
# Debugging Qt Quick Projects

For an example of how to debug Qt Quick Projects, see [Debugging a Qt Quick Example Application](#).

**Note:** In this section, you are using advanced menu items. These are not visible by default. To toggle the visibility of advanced menu items, see [Customizing the Menu](#).

## Setting Up QML Debugging

To debug Qt Quick UI projects, select the **Enable QML** check box in the **Debugger Settings** in **Projects mode Run Settings**.



## Starting QML Debugging

To start the application, choose **Debug > Start Debugging > Start Debugging of Startup Project** or press **F5**. Once the application starts running, it behaves and performs as usual. You can then perform the following tasks:

- Debug JavaScript functions
- Execute JavaScript expressions to get information about the state of the application
- Inspect QML properties and JavaScript variables and change them temporarily at runtime

To debug already running applications:

1. Start the application with the following arguments:

```
-qmljsdebugger=port:<port>[,host:<ip address>][,block]
```

Where **port** (mandatory) specifies the debugging port, **ip address** (optional) specifies the IP address of the host to use for the connection to the target device. For more information, see the [Qt Quick Application Development](#) documentation.

**Note:** Setting breakpoints is only possible if the application is started with block mode.

## 2. Select **Debug** > **Start Debugging** > **Attach to QML Port**.

Choose the kit configured for the device where the application to be debugged is running. The port number to use is displayed in the standard output when the application starts.

# Debugging JavaScript Functions

You can use the Qt Design Studio **Debug** mode to inspect the state of your application while debugging. You can interact with the debugger by:

- › [Setting breakpoints](#)
- › [Viewing call stack trace](#)
- › [Viewing local variables and function parameters](#)
- › [Evaluating Expressions](#)


# Setting Breakpoints


You can associate breakpoints with:

- › Source code files and lines
- › Functions
- › Addresses
- › Throwing and catching exceptions
- › Executing and forking processes
- › Executing some system calls
- › Changes in a block of memory at a particular address when a program is running
- › Emitting QML signals
- › Throwing JavaScript exceptions

The interruption of a program by a breakpoint can be restricted with certain conditions.

Breakpoints come in two varieties: **unclaimed** and **claimed**. An unclaimed breakpoint represents a task to interrupt the debugged program and passes the control to you later. It has two states: **pending** and **implanted**.

Unclaimed breakpoints are stored as a part of a session and exist independently of whether a program is being debugged or not. They are listed in the **Breakpoint Preset** view and in the editor using the  (**Unclaimed Breakpoint**) icon, when they refer to a position in code.

Debuggee	Function	File	Line	Address	Condition	Ignore	Threads
	-	...\quickcontrols2\gallery\gallery.qml	155				(all)

listed in the **breakpoints** view of the running debugger. This view only exists while the debugger is running.

When a debugger claims a breakpoint, the unclaimed breakpoint disappears from the **Breakpoint Preset** view, to appear as a pending breakpoint in the **Breakpoints** view.

At various times, attempts are made to implant pending breakpoints into the debugged process. Successful implantation might create one or more implanted breakpoints, each associated with an actual address in the debugged breakpoint. The implantation might also move a breakpoint marker in the editor from an empty line to the next line for which the actual code was generated, for example. Implanted breakpoint icons don't have the hourglass overlay.

When the debugger ends, its claimed breakpoints, both pending and implanted, will return to the unclaimed state and re-appear in the **Breakpoint Preset** view.

When an implanted breakpoint is hit during the execution of the debugged program, control is passed back to you. You can then examine the state of the interrupted program, or continue execution either line-by-line or continuously.

Number	Function	File	Line	Address	Condition	Ignore	Threads
2	-	f:\dd\vc\tools\crt\vcstartup\src\startup\exe_main.cpp	17	0x7ff7447b9e3d			(all)

## Adding Breakpoints

To add breakpoints:

1. Add a new breakpoint in one of the following ways:
  - › In the code editor, click the left margin or press **F9** (**F8** on macOS) on a particular line you want the program to stop.
  - › In the **Breakpoint Preset** view or the **Breakpoints** view:
    - › Double-click the empty part of the view.
    - › Right-click the view, and select **Add Breakpoint** in the context menu.
2. In the **Breakpoint type** field, select the location in the program code where you want the program to stop. The other options to specify depend on the selected location.

Add Breakpoint

?

×

Basic

Breakpoint type: File Name and Line Number

File name:

Browse...

Line number:

0

Enabled:

☒

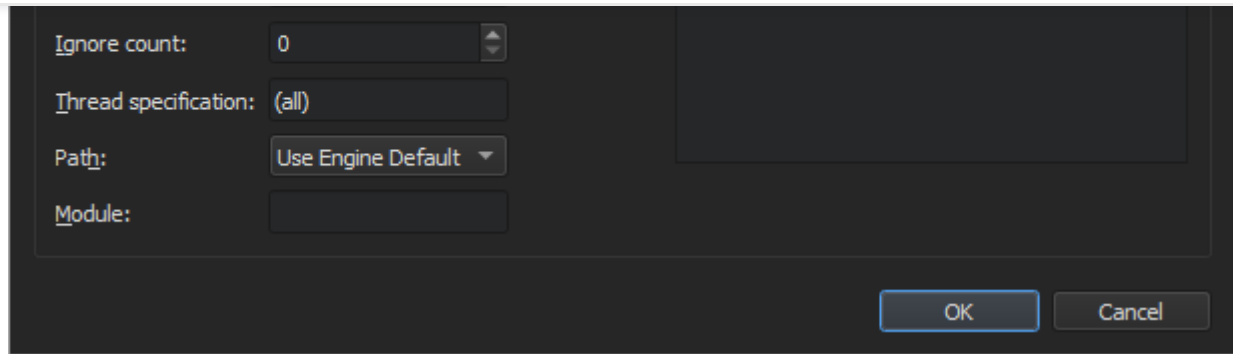
Address:

Expression:

Function:

One shot only:

☐



3. In the **Condition** field, set the condition to be evaluated before stopping at the breakpoint if the condition evaluates as true.
4. In the **Ignore** field, specify the number of times that the breakpoint is ignored before the program stops.
5. In the **Commands** field, specify the commands to execute when the program stops; one command on a line. GDB executes the commands in the order in which they are specified.

## Moving Breakpoints

To move a breakpoint:

- › Drag and drop a breakpoint marker to another line in the text editor.
- › In the **Breakpoint Preset** view or the **Breakpoints** view, select **Edit Selected Breakpoints**, and set the line number in the **Line number** field.

## Deleting Breakpoints

To delete breakpoints:

- › Click the breakpoint marker in the text editor.
- › In the **Breakpoint Preset** view or the **Breakpoints** view:
  - › Select the breakpoint and press **Delete**.
  - › Select **Delete Selected Breakpoints**, **Delete Selected Breakpoints**, or **Delete Breakpoints of File** in the context menu.

## Enabling and Disabling Breakpoints

To temporarily disable a breakpoint without deleting it and losing associated data like conditions and commands:

- › Right-click the breakpoint marker in the text editor and select **Disable Breakpoint**.
- › Select a line that contains a breakpoint and press **Ctrl+F9** (**Ctrl+F8** on macOS).
- › In the **Breakpoint Preset** view or the **Breakpoints** view:
  - › Select the breakpoint and press **Space**.
  - › Select **Disable Breakpoint** in the context menu.

A hollow breakpoint icon in the text editor and the views indicates a disabled breakpoint. To re-enable a breakpoint, use any of the above methods.

With the notable exception of data breakpoints, breakpoints retain their enabled or disabled state when the debugged program is restarted.

A **data breakpoint** stops the program when data is read or written at the specified address.

To set a data breakpoint at an address:



1. In the **Breakpoint Preset** or **Breakpoints** view, select **Add Breakpoint** in the context menu.
2. In the **Breakpoint type** field, select **Break on data access at fixed address**.
3. In the **Address** field, specify the address of the memory block.
4. Select **OK**.

If the address is displayed in the **Locals** or **Expressions** view, you can select **Add Data Breakpoint at Object's Address** in the context menu to set the data breakpoint.

Data breakpoints will be disabled when the debugged program exits, as it is unlikely that the used addresses will stay the same at the next program launch. If you really want a data breakpoint to be active again, re-enable it manually.

## Viewing Call Stack Trace

When the program being debugged is interrupted, Qt Design Studio displays the nested function calls leading to the current position as a call stack trace. This stack trace is built up from call stack frames, each representing a particular function. For each function, Qt Design Studio tries to retrieve the file name and line number of the corresponding source file. This data is shown in the **Stack** view.

Stack  			
Level	Function	File	Line
➔ 1	TextFinder::loadTextFile	textfinder.cpp	85
2	TextFinder::TextFinder	textfinder.cpp	64
3	main	main.cpp	59
4	invoke_main	exe_common.inl	79
5	__scrt_common_main_seh	exe_common.inl	283
6	__scrt_common_main	exe_common.inl	326
7	mainCRTStartup	exe_main.cpp	17
8	BaseThreadInitThunk	KERNEL32	
9	RtlUserThreadStart	ntdll	

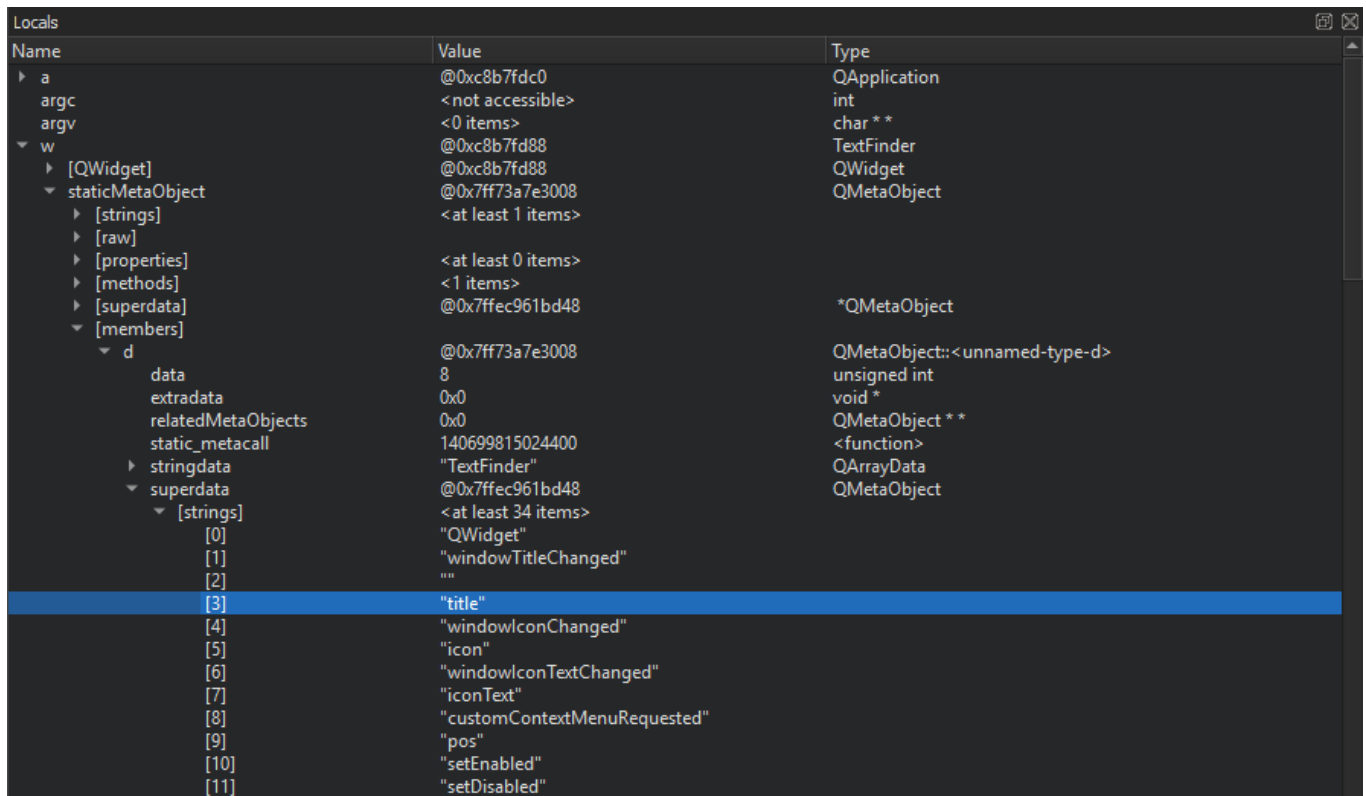
Since the call stack leading to the current position may originate or go through code for which no debug information is available, not all stack frames have corresponding source locations. Stack frames without corresponding source locations are grayed out in the **Stack** view.

If you click a frame with a known source location, the text editor jumps to the corresponding location and updates

in the **Stack** view. The debugger tries to retrieve the JavaScript stack from the stopped executable and prepends the frames to the C++ frames, should it find any. You can click a frame in the QML stack to open the QML file in the editor.

## Local Variables and Function Parameters

The Locals view consists of the **Locals** pane and the **Return Value** pane (hidden when empty).

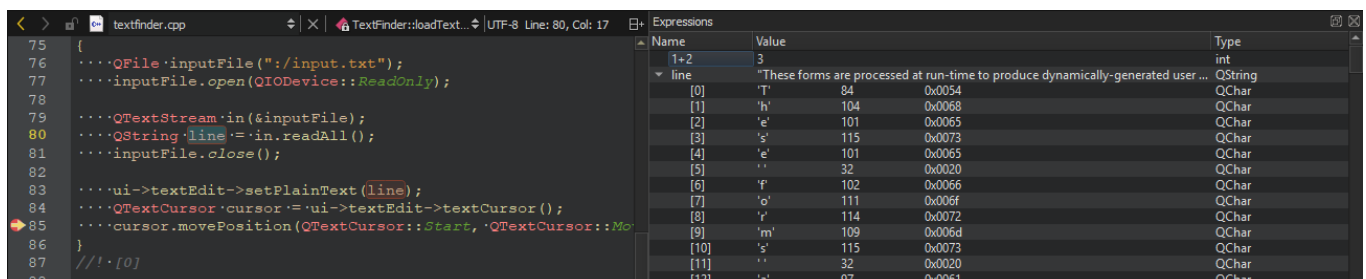


Whenever a program stops under the control of the debugger, it retrieves information about the topmost stack frame and displays it in the **Locals** view. The **Locals** pane shows information about parameters of the function in that frame as well as the local variables. If the last operation in the debugger was returning from a function after pressing **Shift+F11**, the **Return Value** pane displays the value returned by the function.

When using GDB, you can specify whether the dynamic or the static type of objects will be displayed. Select **Use dynamic object type for display** in the context menu. Keep in mind that choosing the dynamic type might be slower.

## Evaluating Expressions

To compute values of arithmetic expressions or function calls, use expression evaluators in the **Expressions** view. To insert a new expression evaluator, either double-click on an empty part of the **Expressions** or **Locals** view, or select **Add New Expression Evaluator** from the context menu, or drag and drop an expression from the code editor.



**Note:** Expression evaluators are powerful, but slow down debugger operation significantly. It is advisable to not use them excessively, and to remove unneeded expression evaluators as soon as possible.

Expression evaluators are re-evaluated whenever the current frame changes. Note that functions used in the expressions are called each time, even if they have side-effects.

The QML debugger can evaluate JavaScript expressions.

## Inspecting Items

While the application is running, you can use the **Locals** view to explore the QML item structure.

Locals			
Name	Value	Type	
QQuickView	object	QQuickView	
Properties	list		
QQuickRootItem	object	QQuickRootItem	
Properties	list		
root	object	Rectangle	
Properties	list		
Transition	object	Transition	
State	object	State	
Properties	list		
changes	<unknown valu...	QQmlListProperty<QQuickStateOperation>	
extend		QString	
name	in-game	QString	
objectName		QString	
when		QQmlBinding *	
gameOverTimer	object	Timer	
Image	object	Image	
gameCanvas	object	GameArea	
menu	object	Item	
scoreBar	object	Image	
bottomBar	object	Image	
Connections	object	Connections	
stateChangeAnim	object	SequentialAnimation	
Keys	object	Keys	
QQmlEngine	object	QQmlEngine	
QQmlFileSelector	object	QQmlFileSelector	
Component	object	Component	

To keep the application visible while you interact with the debugger, select **Debug > Show Application on Top**.

You can view a QML item in the **Locals** view in the following ways:

- Expand the item in the object tree.
- Select the item in the code editor.
- Select **Debug > Select** to activate selection mode and then click an item in the running application.

To change property values temporarily, without editing the source, double-click them and enter the new values. You can view the results in the running application.

## Inspecting User Interfaces

In the selection mode, you can click items in the running application to jump to their definitions in the code. The properties of the selected item are displayed in the **Locals** view.

You can also view the item hierarchy in the running application:

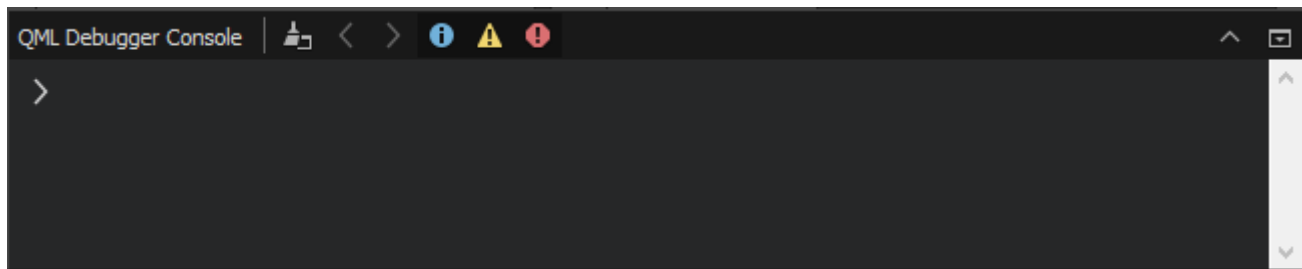
Double-click an item in the running application to cycle through the item stack at the cursor position.

To switch out of the selection mode, toggle the **Select** menu item.

To move the application running in Qt QML Viewer to the front, select **Debug > Show Application on Top**.

## Executing JavaScript Expressions

When the application is interrupted by a breakpoint, you can use the **QML Debugger Console** to execute JavaScript expressions in the current context. To open it, choose **View > Output > QML Debugger Console**.



You can change property values temporarily, without editing the source, and view the results in the running application. You can change the property values permanently in the **Properties** view.

## Applying QML Changes at Runtime

When you change property values in the **QML Debugger Console** or in the **Locals** or **Expression** view, they are immediately updated in the running application, but not in the source code.

[< Debugging and Profiling](#)

[Debugging a Qt Quick Example Application >](#)



Contact Us

Company

Licensing





Newsroom

Careers

Office Locations

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