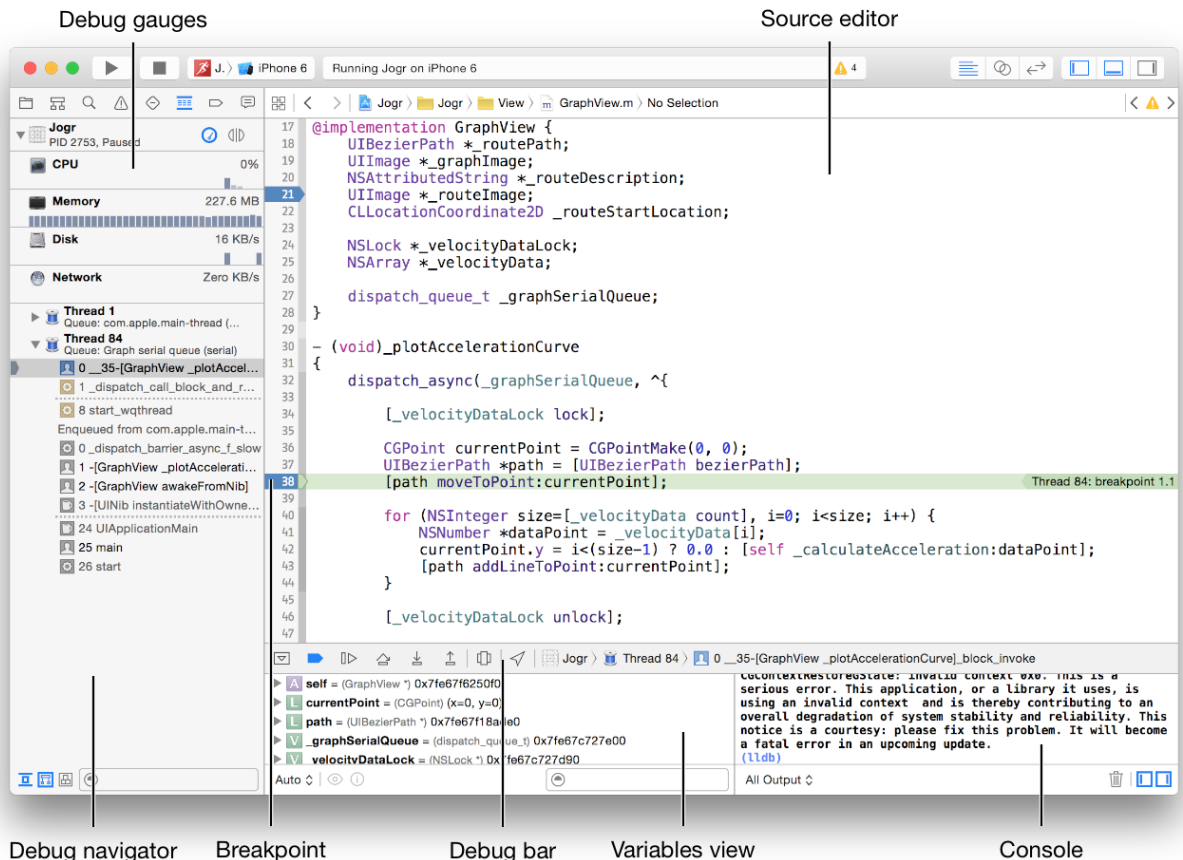


Debugging Tools

The Xcode debugging tools are integrated throughout the Xcode main window but are primarily located in the Debug area, the debug navigator, the breakpoint navigator, and the source editor. The debugging UI is dynamic; it reconfigures as you build and run your app. To customize how Xcode displays portions of the UI, choose Xcode Preferences > Behaviors.

The illustration below shows the default layout of the Xcode debugger with the app paused at a breakpoint.



General Notes

Here are a few notes about debugging in general and some basic information about Xcode as you begin to read this chapter.

The Five Parts of Debugging and the Debugging Tools

There are five parts to the debugging workflow:

- **Discover.** Identify a problem.
- **Locate.** Determine where in the code the problem occurs.
- **Inspect.** Examine the control flow and data structures of the running code to find the cause of the problem.
- **Fix.** Apply your insight into the cause of the problem to devise a solution, and edit the code to suit.
- **Confirm.** After editing, run the app and check it with the debugger to be sure the fix was successful.

The division of labor in these five parts of debugging are not necessarily reflected in the specifics of the debugging tools, although some tools are more pointed at discovery (for instance, the debug gauges), some are particularly useful for dealing with locations of interest in your code (breakpoints), and others are more specific to inspection (the debug area's variables view and the debug navigator's process view).

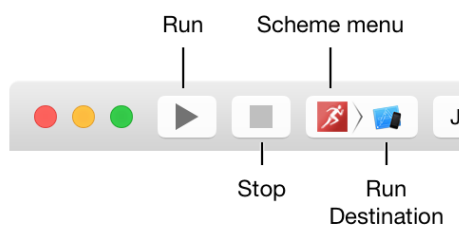
Think of the debugging tools as addressing the five parts of debugging more as a function of how you use them. For instance, you use the Quick Look feature to inspect the state of a graphical object as you work through a problem, but you can also think of it as being a discovery tool, using it to see how a complex graphic is “assembled” as you iterate through a set of drawing calls. In other words, how you put a tool to work for you often determines which part of the debugging effort it addresses, and it is the combination of what the tool does, what data using it uncovers, and your creative insight into the context of the situation that delivers success to your debugging efforts.

LLDB and the Xcode Debugger

The Xcode debugger uses services and functions provided by LLDB, the underlying command-line debugger that is part of the LLVM compiler development suite. LLDB is tightly integrated with the compiler, which enables it to provide the Xcode debugger with deep capabilities in a user-friendly environment. The Xcode debugger provides all the functionality needed for most debugging situations, but a little familiarity with LLDB can be helpful. For a basic introduction to LLDB, see *LLDB Quick Start Guide*.

Xcode Toolbar Controls

The Xcode toolbar contains the most basic controls you need to start debugging.



Run button. Click to build and run. Click and hold to select other actions (Run, Test, Profile, Analyze) from a menu. Using the Shift key modifies the menu to a “Build for” operation; similarly, using the Control key modifies the menu to perform an action “without Building.” The default operation is to build and run, which starts the debugger as well.

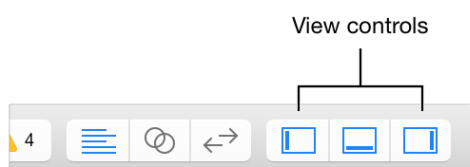
Note: The Run button presents an alternative way of choosing actions from the Product menu.

Stop button. Click to stop the current running task or app.

Scheme menu. Xcode schemes control the build process based on the settings they contain for the Product action you choose and the target build settings. For most uses, the defaults created with a project suffice, but there are useful debugging options configurable in the scheme editor’s Run action. A look at these options is provided in Debugging Options in the Scheme Editor.

Run destination. Choose from this menu the macOS or iOS device (or simulator) the build and run operation will execute on.

Three buttons control the visibility of the navigator area, debug area, and utility area. These buttons, known as view controls, have matching commands with keyboard shortcuts in the View menu.

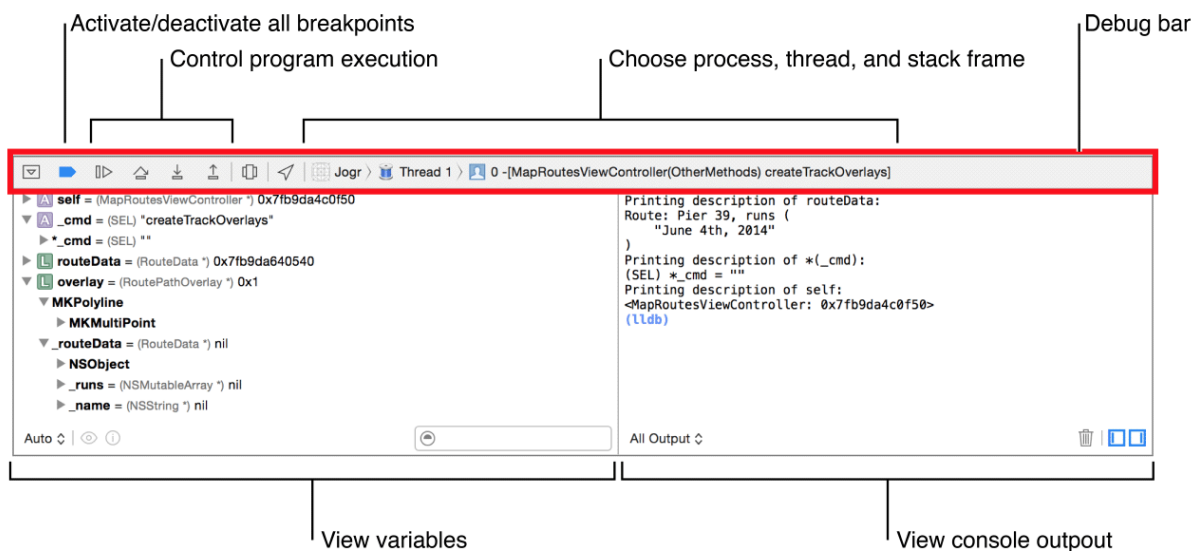


Xcode Debug and Product Menus

In addition to the areas and controls located in the Xcode main window panels and toolbar, the Xcode main menu bar includes the Product and Debug menus. These menus provide a convenient, configurable keyboard mapping for many of the more common commands used during debugging sessions. The Debug menu provides convenient keyboard stepping commands for use when you pause your app and analyze what happens line by line. It also gives you access to some of the less-used debugging functions, such as the `Debug > Attach to Process` command, which allows you to target an app that is already running with the debugger. As you become proficient with the Xcode debugger, you'll likely make more use of these menu shortcuts.

Debug Area

The debug area opens at the bottom of the Xcode main window below the source editor when you build and run your app. It contains the debug bar, the variables view, and the console.

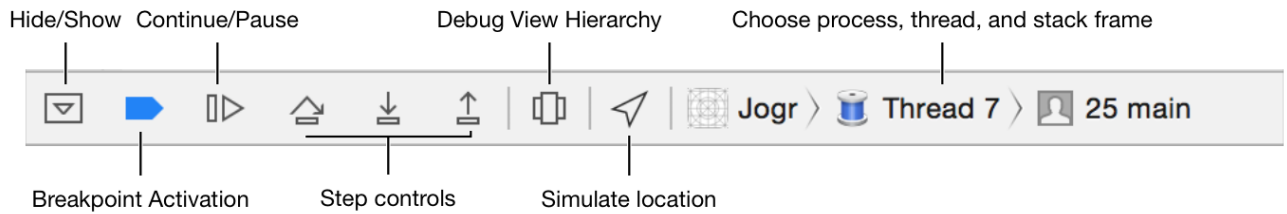


To hide or show the debug area click the center button  in the view controls group located in the main window toolbar.

Debug Bar–Process Controls



When you build and run a program in Xcode, the debug bar appears at the bottom of the editor pane. The debug bar includes buttons to:




- Open or close the debug area
- Activate or deactivate all breakpoints
- Pause or resume execution of your code
- Step over; that is, execute the current line of code and, if the current line is a function or method, step out to the next line in the current file
- Step in; that is, execute the current line of code and, if the current line is a routine, jump to the first line of that routine
- Step out of a jumped-to routine; that is, complete the current routine and step to the next routine or back to the calling routine.



Debug area Hide/Show button. The action of this button differs from the view controls in the Xcode main window toolbar in that when you run a debugging session, it controls the view of both variable view and console panels but leaves the debug bar accessible. This is useful when you are working primarily in the source editor while debugging and want to maximize the amount of space you have to view your source code.

Breakpoint Activation button. This button acts as a toggle to deactivate and activate all breakpoints in your app simultaneously. It's useful when you know you need to let the app run normally, without pausing at any breakpoints, for a while to reach a state where you can start debugging a problem.

Continue/Pause button. You can suspend the execution of your app by clicking the Pause button, which toggles between  to pause and  to continue. More commonly, however, you set a breakpoint to pause your app at a planned location.

Step controls. When your app is paused, the currently executing line of code is highlighted in green. You can step through execution of your code using step over () , step into () , and step out () . Step over will execute the current line of code, including any methods. If the current line of code calls a method, step into starts execution at the current line, and then stops when it reaches the first line of the called method. Step out executes the rest of the current method or function.

The stepping controls have alternative operations for working with disassembly or threads. You use the Control and Control-Shift modifier keys to call these alternatives. Press Control to step by assembly language instruction instead of by statement (the step icons change to show a dot rather than a line under the arrow) or Control-Shift to step into or over the active thread only while holding other threads stopped (the step icons show a dashed rather than solid line below the arrow).

Debug View Hierarchy button. Click to investigate the relationship of view objects both in a 3D rendering and in a hierarchical list in the debug navigator. See Debugging the View Hierarchy for more information.

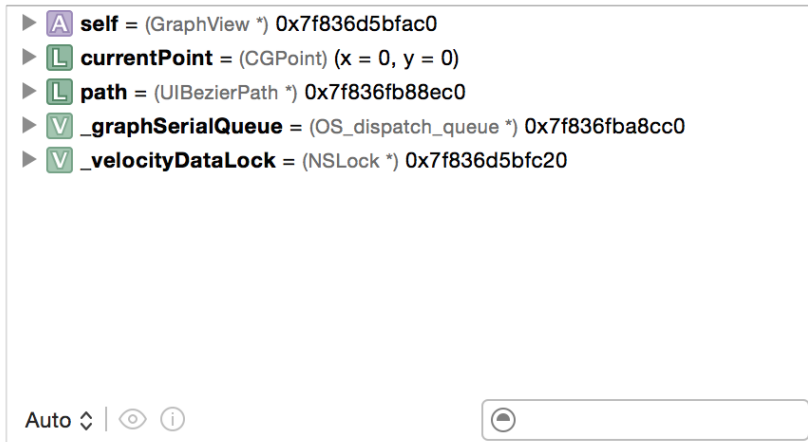
Simulate Location button. Clicking this button presents a menu of locations to choose from. You choose one to tell a device or the simulator to report that location to your app as it runs.

Process/Thread/Stack frame jump bar. When your app is paused, this jump bar provides a convenient way to navigate through all processes and threads, and lets you jump to specific stack frames for the purpose of inspecting the program flow or setting breakpoints. Picking a stack frame from the jump bar will bring the source, or disassembly if the source file isn't available, into the source editor.

Variable View—Inspecting Variables

The variable view provides the primary way to inspect the state of variables when an app is paused.

Variable list. The variable view lists each variable available in the context where the app is paused. Each variable at the top level takes up a row in the list. Disclosure triangles are at the far left of the row, followed by an identifying icon, the variable name, type, and value.

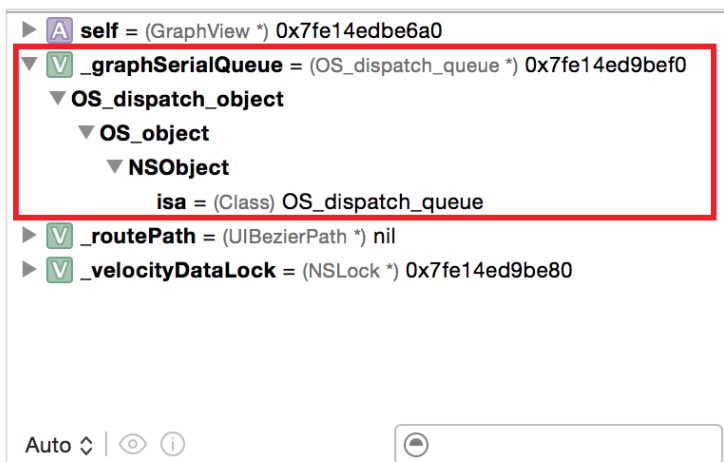


The icons used in the variable list allow easy recognition of the variable kind at a glance.

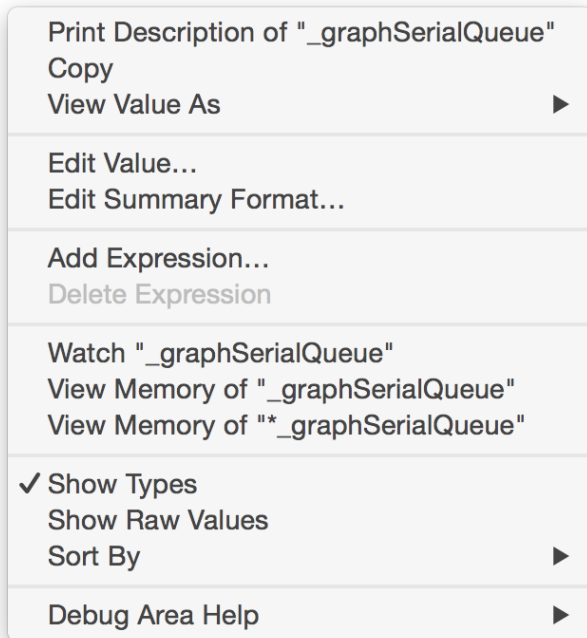


The variable name is followed by the variable value as generated by the data formatter for the variable type. As you use the stepping controls in the debug bar to execute your app line by line, you can see the variable values that result from each operation in the source.

Clicking the disclosure triangle on the left, you “open up” the variable to inspect its component parts and their values, as in the red boxed variable below.



Variable view menu. Right- or Control-click a variable in the list to display commands that act on the variable. Some of the commands duplicate other ways to obtain the same behavior.



Print Description of “{variable}”. An equivalent to using `po` in the console or using the Print Description button in a tool tip.

Copy. Copies the selected variable’s representation displayed in variable view, allowing you to paste it as text.

View Value As. You can use this command to cast a variable’s type to another type and apply that data formatter in variable view for display. Clicking on the command presents a menu of standard types. The menu includes a Custom Type option, which presents a pop-up editor that you use to input a custom data format.

Edit Value. This command puts the selected variable’s value into an edit field so you can type in a specific value. Alternatively, you can double-click the variable value in the list directly to achieve the same function.

Edit Summary Format. Presents an editor allowing you to change the data formatter by entering a new formatter representation for the variable.

Add Expression. Allows you to add an expression to the variable list for the debugger to evaluate and present a result in variable view. An option in the editor allows you to add the expression to the variable list in all stack frames.

Watch “{variable}”. Creates a watchpoint that reports the value of a variable as your app runs. Watchpoints are managed in the breakpoint navigator.

View Memory of “{variable}”. Using this command changes the display in the source editor to a hex editor display based on the address of `{variable}`.

View Memory of “*{variable}”. Similar in function to the preceding, this command changes the display in the source editor to a hex editor display based on the address pointed to by `{variable}`.

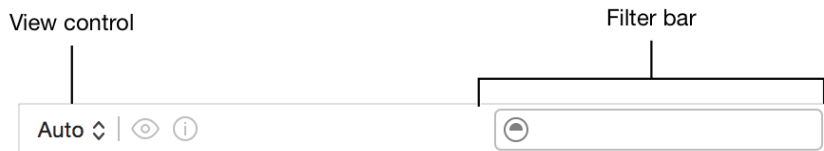
Note: For both of these View Memory modes, return to the source listing by pressing the back button, by choosing the stack frame in the debug bar’s jump bar, by clicking on the stack frame in the debug navigator, or by clicking the breakpoint in the breakpoint navigator.

Show Types and Show Raw Values. Allow you to adjust how much information is presented for each variable in the variable view list.

Sort By. Allows you to choose to show variables by their order of appearance in the source or by name, in ascending or descending order (when showing by name).

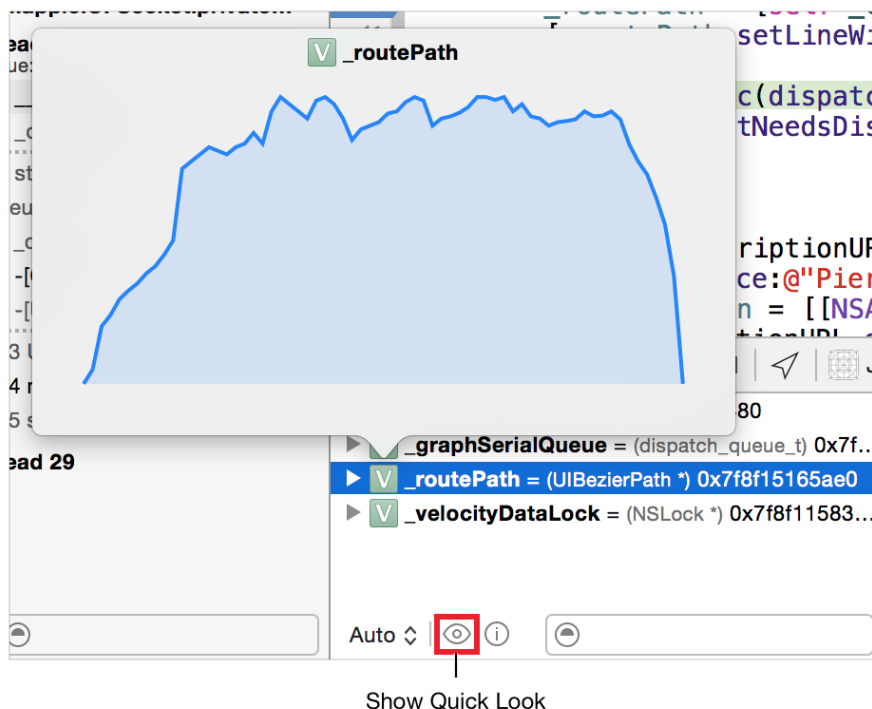
View control menu. The variable view can display both variables and registers. You specify which items to display using the pop-up menu in the lower-left corner. By default, it is set to Auto. The options are:

- **Auto** displays only the variables you’re most likely to be interested in, given the current context.
- **Local** displays local variables.
- **All** displays all variables and registers.



Filter bar. Use the search field to filter the items displayed in the variables pane.

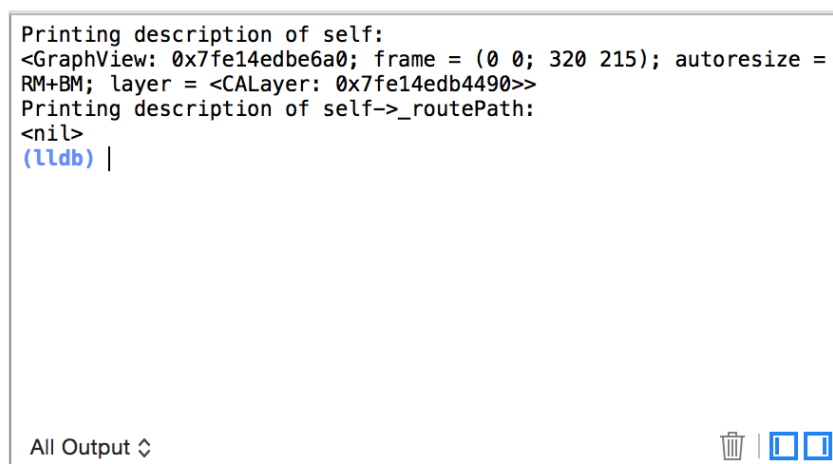
Quick Look button. Use Quick Look to produce a visual rendering depending upon the type of the selected variable. Quick Look graphical rendering is particularly useful when you are trying to see a complex object and how it is being drawn or rendered. For example, this illustration shows a `UIBezierPath` object in the variables view:



Print Description button. Select a variable in the list and click the Print Description button ⓘ to print textual information about the variable to the console pane. It is equivalent to using the LLDB `po` command.

Console—Command-Line Input/Output

The debug area console is a command line environment that you can use to interact with an app or with LLDB.



Command-line environment. When the app is running during a debugging session, it can read from the console using `stdin` and output to the console using `stdout` and `stderr`. Logging with `NSLog` is sent to the console as well. When an app is paused, input typed in the console is interpreted by LLDB; output from LLDB is sent to the console as well.

Note: Running an app within the Xcode environment but without attaching to the debugger is also possible, in which case the console is still accessible to an app as `stdin`, `stdout`, and `stderr`.

Console output persists throughout a debugging session, you can see a listing of outputs created for the entire session. When you finish a session and rerun the app, the console is cleared. You can review the console contents from previous debugging sessions by going to the Report navigator and checking the contents of the debug sessions stored there.

Output control. You specify the type of output the console displays with the pop-up menu in the lower-left corner of the console pane:

- **All Output** displays target and debugger output.
- **Debugger Output** displays debugger output only.
- **Target Output** displays target output only.

All output is the console default.

Clear console button. The Trash button at the lower right allows you to clear the console at any time in your session. (The full console output is logged to the Reports navigator.)

Debug area view controls. Click the buttons at the lower right of the console to control the display of the debug area. They allow you to hide or show either the variable view or the console.

Breakpoints and the Breakpoint Navigator

Breakpoints

A breakpoint is a mechanism to pause an app during execution at a predetermined location in order to inspect the state of variables and the app's control flow. There are several different kinds of breakpoints, and they can be edited to add conditions, actions, and scripts. Although you can use the debugger to pause an app at any time, it's helpful to set breakpoints before and even while your app is running so that you can pause it at known points where you have determined problems might be occurring.

Kinds of breakpoints. The most commonly used breakpoint is file and line dependent: After you create and enable a file and line breakpoint, your app pauses every time it encounters that location in the code. You can set several options for a breakpoint, such as a condition, the number of times to pass the breakpoint before it's triggered, or an action to perform when the breakpoint is triggered. Conditions and actions are the most commonly used breakpoint options.

In addition to file and line breakpoints, you can create exception breakpoints, which are triggered when a specific type of exception is thrown or caught, and symbolic breakpoints, which are triggered when a specific method or function is called.

A test failure breakpoint is a specialized type of breakpoint used in debugging Xcode tests; see *Testing with Xcode* for details on its creation and use.

An OpenGL ES error breakpoint is a specialized type of breakpoint used in the Open GL ES debugging tools, it is a derivative of a symbolic breakpoint. See *OpenGL ES Programming Guide for iOS* for details on its creation and use.

You can also create watchpoints. Watchpoints are breakpoints that don't stop execution, they report the state of the variable they're configured for each time that variable is used. Set a watchpoint by Control-clicking the variable in the debug area variable view and choosing "Watch {variable name}" from the pop-up menu. There are a limited number of watchpoints available depending upon the hardware the app is running on.

Adding and enabling breakpoints. The most commonly used file and line breakpoints for methods and functions are created in the source editor; see Source Editor for details. Exception and symbolic breakpoints

are created with the Add (+) button at the bottom of the breakpoint navigator by choosing Add Exception Breakpoint or Add Symbolic Breakpoint from the pop-up menu. An editor is displayed that allows you to set the parameters for these types of breakpoints.

Editing breakpoints. To set breakpoint options, Control-click the breakpoint for which you want to set options, either in the breakpoint navigator or the source editor, then choose Edit Breakpoint from the shortcut menu. The Condition field lets you specify an execute condition for the breakpoint using an *expression*.

An expression can be written in Objective-C or Swift, and can include function calls. You can use any variables that are in the current scope for that breakpoint. When you specify a condition, the breakpoint is triggered only if the condition (for example, `i == 24`) evaluates to `true`.

Condition field expressions—function return types:

Expressions used in the breakpoint editor, just like expressions used with the LLDB expression parser, need to be the correct return type before the expression parser can evaluate them. For Objective-C, debug information often supplies this information automatically; in other cases Xcode can obtain that information from the Objective-C runtime for you. Otherwise you need to cast the function return type in the expression.

For Swift, since you can overload function return types, the return type is a part of the mangled name and is generally available even without debug information, as long as the Swift modules are available.

A good strategy to use when constructing conditional expressions to use is to first stop the app at the breakpoint unconditionally and then try your conditional expression in LLDB using the console. If it works, you can put it into the condition and you're set. If your attempt doesn't work, the compiler diagnostics are visible—you use them to fix up the expression for use.

Breakpoint behavior. What Xcode does when a breakpoint is encountered and your app is paused can be configured by the settings in Xcode Preferences > Behaviors using the Running > Pauses options.

Breakpoint scope. Breakpoints have scope; that is, they have a context in which they are defined and operate. By default, a new breakpoint is local to the workspace you have open. In this case, if you add the project containing that breakpoint to another workspace, the breakpoint is not copied to the new workspace.

You can assign breakpoints to two other scopes:

- A breakpoint with User scope appears in all of your projects and workspaces, wherever that source file is used.
- A breakpoint set to a specific project can be seen whenever you open that project, regardless of which workspace the project is in.

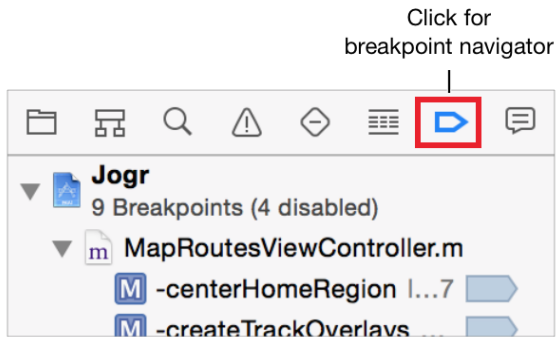
Note: The default scope changes to the project if the project is not part of a workspace.

Breakpoints can be shared with other users. To share a breakpoint with other users, in the breakpoint navigator select the breakpoint you want to share, Control-click the breakpoint, and choose Share Breakpoint from the shortcut menu. Xcode moves shared breakpoints into their own category in the breakpoint navigator; all users with access to the project will be able to use them.

To change the scope of a breakpoint, use the breakpoint navigator. Control-click the breakpoint and choose the scope from the Move Breakpoint To menu item. The scopes of breakpoints are shown in the breakpoint navigator.

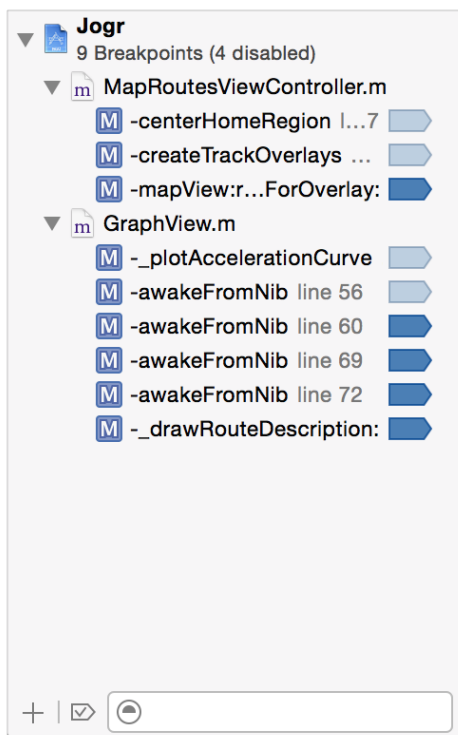
Breakpoint Navigator

The breakpoint navigator helps you manage the breakpoints in your projects and workspaces. Open the breakpoint navigator by clicking the button in the navigator bar.

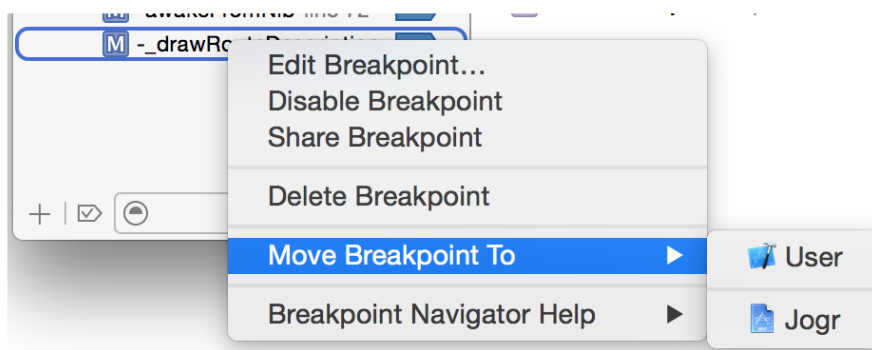


Hierarchical list organization. Breakpoints are organized in a hierarchy by scope, then by containing file. The list includes the symbol name of the function or method that a breakpoint is contained in as well as the line number in the source file.

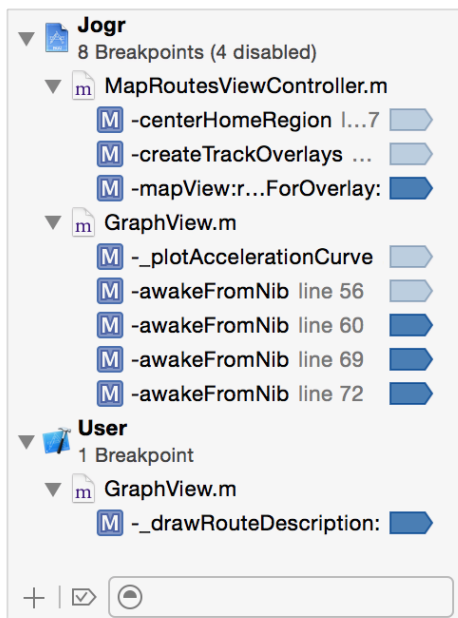
A breakpoint indicator symbol to the right of a breakpoint location—the same as the breakpoint indicators used in the source editor—is either dark or light blue to show whether the breakpoint is enable or disabled, respectively. This example shows nine breakpoints defined in two files:



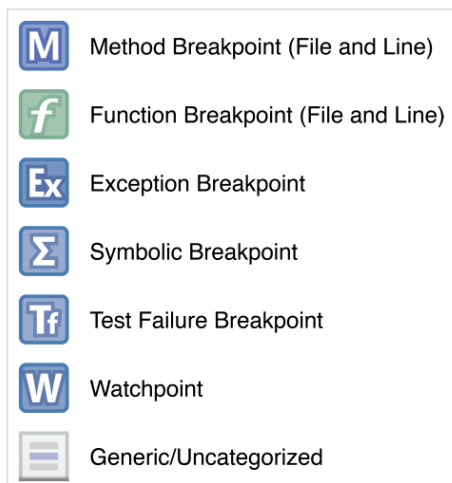
Because these breakpoints were created and left at the defaults, they are all set to project scope. To change the scope of a breakpoint, Control-click the breakpoint and use Move Breakpoint To > User or Move Breakpoint To > Jogr, the two available choices since this is a project not contained in a workspace.



The result of that change is shown in the hierarchy like this:



Just like variables listed in variable view, the several kinds of breakpoints listed in the breakpoint navigator are tagged with icons to make it easy to recognize them.



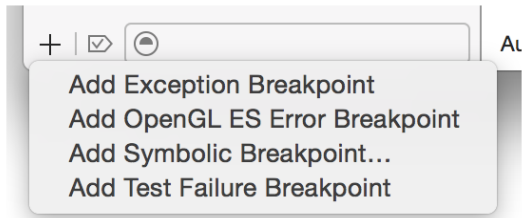
Note: Xcode categorizes breakpoints that you create in the source editor gutter as “method” or “function” based on the context of the source code they are a part of. A breakpoint placed at an inappropriate location is categorized as “generic/uncategorized”—these are usually unreachable points in the source code file, such as between functions or methods, placed by accident.


Disable/enable breakpoints. You click on the breakpoint indicators individually to disable or reenble them. The color changes from dark to light when disabled to indicate the state.

Note: Clicking the Breakpoint Activation button in the debug bar globally toggles all breakpoints in all files and scopes to be activated or deactivated per their individual enabled/disabled state. When deactivated globally, the indicators in the breakpoint navigator are dark and light gray, reflecting the individual state of each breakpoint.

Breakpoint editing. Control-click the breakpoint in the breakpoint navigator and choose Edit Breakpoint from the pop-up menu to display the breakpoint editor. Choosing Edit Breakpoint displays the breakpoint editor, allowing you to set conditions, add actions, and so forth, as mentioned in Breakpoints.

Filter bar. To add an exception, symbolic, or other type of breakpoint to your code, click the Add (+) button at the lower left.

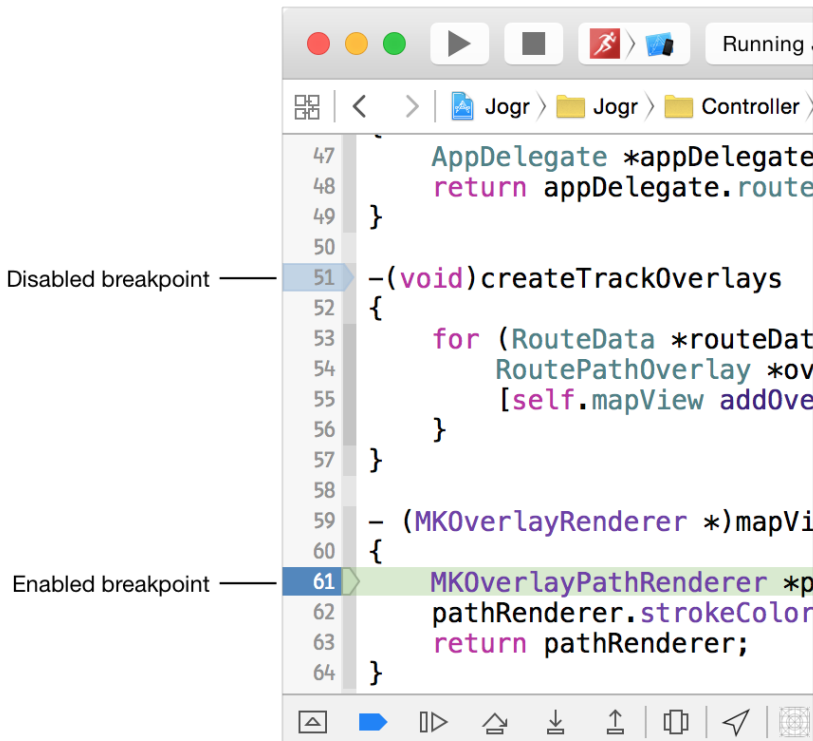


Click the “enabled breakpoints” filter button  to show only the enabled breakpoints in your project. Enabling this filter is handy if you have a lot of breakpoints defined but are working with only a small set of them in an enabled state.

Type into the text filter to limit the view of breakpoints in all scopes and files to ones with text that matches. Using the text filter is a fast way to find a breakpoint in a function or method when you have a large project with many files and defined breakpoints.

Source Editor

In addition to displaying and editing your source code, the source editor plays a vital role in debugging operations. You set breakpoints in the source editor, and you often manage them dynamically during a debugging session. Having your code first and foremost in front of you can also be the best way to work through a problem—you can minimize the debug area and debug navigator to use the source editor as the debugging interface.



The illustration above shows the source editor, with debug area and debug navigator collapsed, and the app paused by an enabled breakpoint. The green pointer and highlight on line 61 in the source indicates the location of the instruction pointer and the current line of code to be executed.

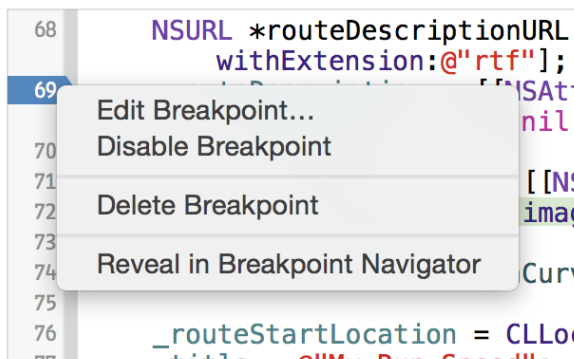
Note: You can manipulate the instruction pointer in the source editor. Control-click in the gutter (not on a breakpoint), choose “Continue to Here,” and Xcode executes the source code up to that point. This

approach is often useful if your code needs to execute an extensive loop from where you wanted to break for inspection before reaching the next point that you want to inspect again.

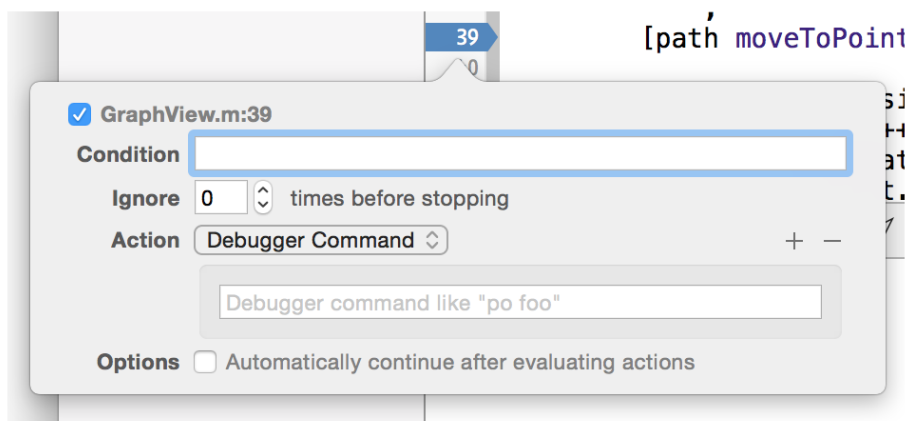
You can similarly drag the instruction pointer to a location; Xcode warns you that moving the instruction pointer this way can cause problems up to and including a crash. What this operation does is simply move the instruction pointer to a different location, enabling you to start from a different location. This kind of instruction pointer manipulation is only infrequently useful for testing very specific situations, for example, when you've accidentally stepped over a line in the source and need to force it to be executed.

Create/delete breakpoints. Click the gutter next to the line where you want execution to pause. When you add a breakpoint, Xcode automatically enables it. When an enabled breakpoint is encountered during execution, the app pauses and a green indicator shows the position of the program counter. You can create breakpoints at any time, including when your app is running in a debugging session. If a breakpoint is not in the ideal location, drag the indicator to a better location. To delete a breakpoint, drag the indicator out of the gutter.

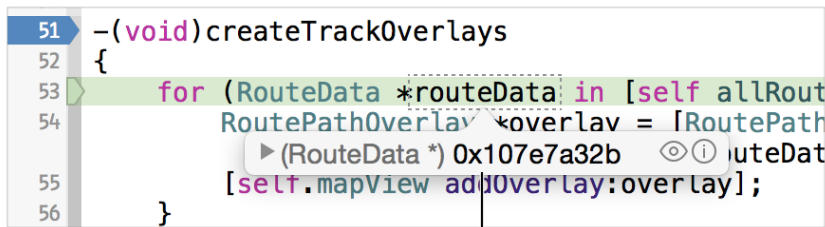
Disable/enable breakpoints. You disable a breakpoint in the source editor by clicking the breakpoint indicator, even while your app is running; this allows you to adjust set locations where the app will be paused as you work through a problem. Disabled breakpoints display a dimmed blue indicator. Enable a breakpoint by clicking the indicator again.



Edit breakpoints. Control-click a breakpoint indicator to display a command menu and choose Edit Breakpoint to open the breakpoint editor and set conditions, add actions, and so forth, as mentioned in Breakpoints.



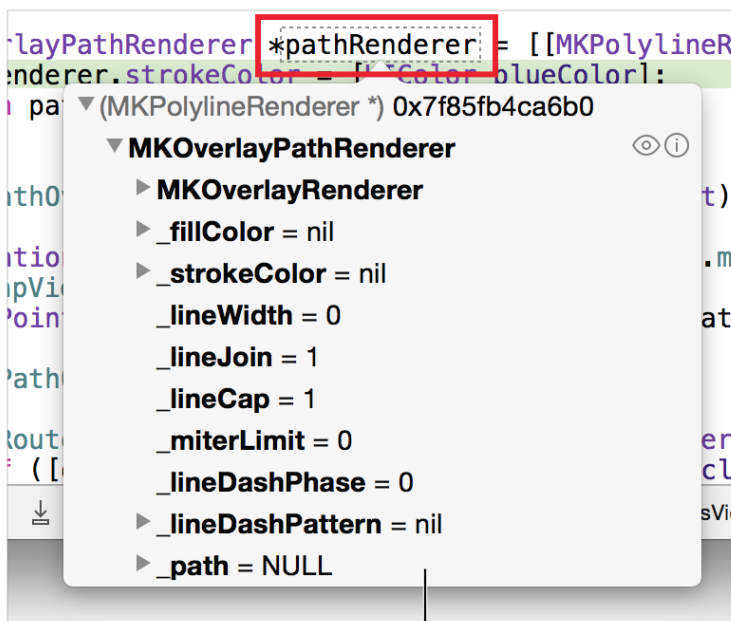
Tip: Option-Command-click on a breakpoint to open the breakpoint editor without having to choose Edit Breakpoint from a menu.



Datatip showing
'routeData' variable

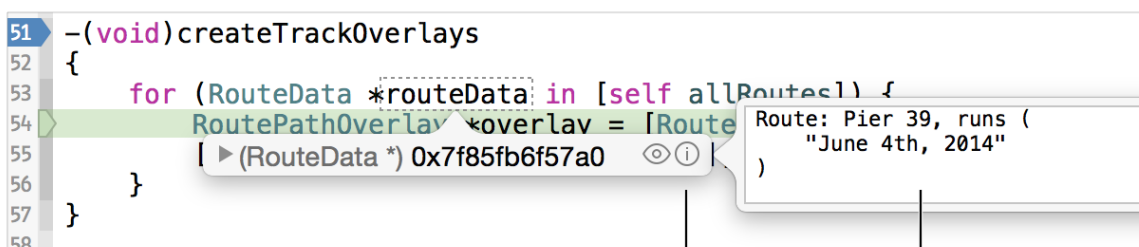
Datatips. You can use the source editor alone during a debugging session with the debug area hidden, showing only the debug bar, to give you a larger view of your source code and the flow of your program as you step.

You can inspect variables directly in the source editor with a datatips pop-up window, which shares the same display layout as the debug area variable view. Both the Quick Look and Print Description buttons are immediately available from the datatips pop-up window, and operate entirely within the source editor. For example, the following illustration shows a datatip open on the pathRenderer variable, outlined at the point of the popup window with a dotted rectangle.



Datatip with
'pathRenderer' disclosed

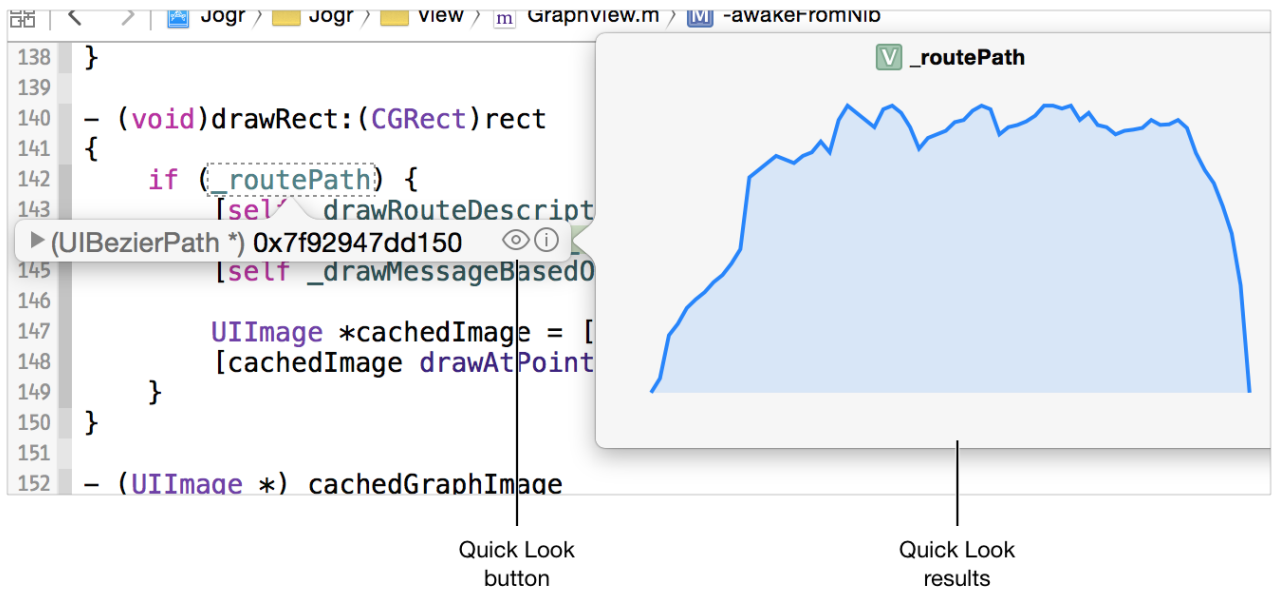
Print Description button. In this example, you hold the mouse over the routeData variable to display a datatip. When the Print Description button is clicked, results are displayed just as they would be in the console in a popover window.



Print Description
button

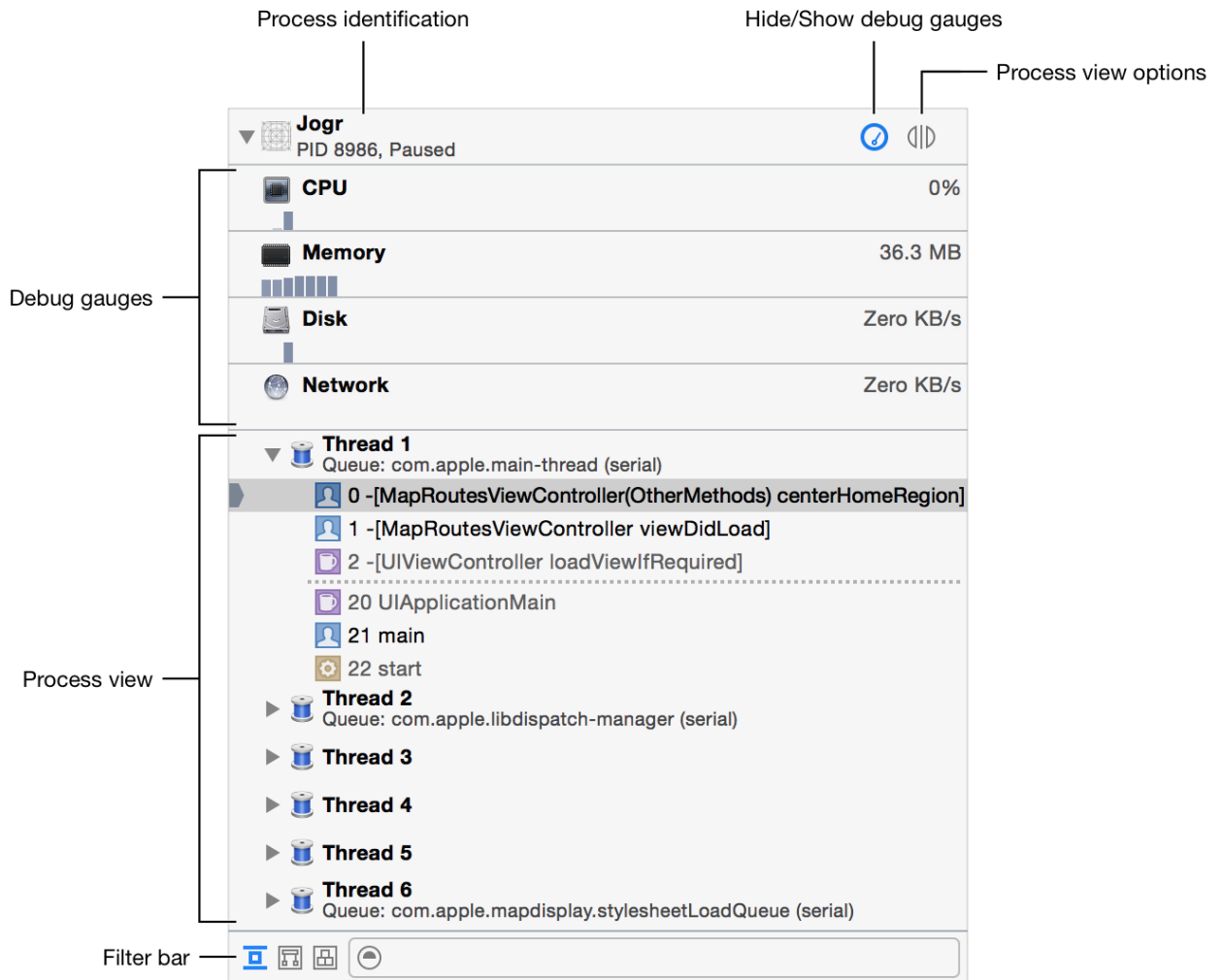
Print Description
results

Quick Look button. The Quick Look feature works similarly, this time electing the `_routeProvider` variable in the example.



The Debug Navigator

The debug navigator has two main parts, debug gauges and process view display, as well as view controls and a filter bar. It provides tools for discovering issues in your running app and presents the call stack for inspection when the app is paused.



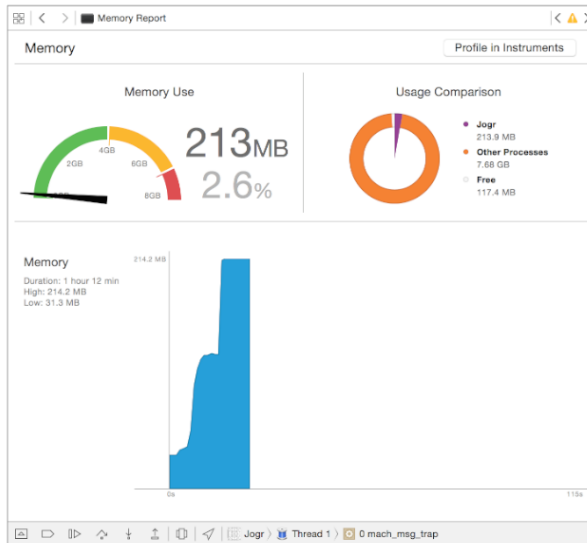
Debug Gauges

The debug gauges monitor your running app. Debug gauges constantly change as your app runs—they are presented as a histogram running from left to right in time. Spikes in the histogram show activity.


There are seven types of debug gauges: CPU, Memory, Energy, Disk I/O, Network I/O, GPU, and iCloud. They are displayed depending on platform and app capabilities. CPU, Memory, Disk I/O, and Network I/O are available for all app development. The Energy gauge is available in macOS apps when running Xcode 6.3 and greater, and is available for iOS apps with Xcode 7 or greater. The iCloud and GPU gauges are available when your app target has been subscribed to associated capabilities with the project editor.

The debug gauges provide insight into how your app is performing in its use of system resources. Depending on the capabilities of your app and the characteristics of its destination, gauges can report your app's impact on the system and other running apps. Observe the debug gauges for a running app to become familiar with the gauges' normal variation and with the standard behaviors of the app over time. When you make changes to the app, look for differences in the debug gauge's readings and look into any new behavior that seems anomalous or indicative of a problem compared to the app's previous running behavior.

Click a debug gauge as your app runs to display a live *detail report* in the main editor. The detail report of each type of gauge differs in the specifics, but all follow a similar pattern. As an example, here is a detail report shown when you click the Memory debug gauge, captured from a paused app:

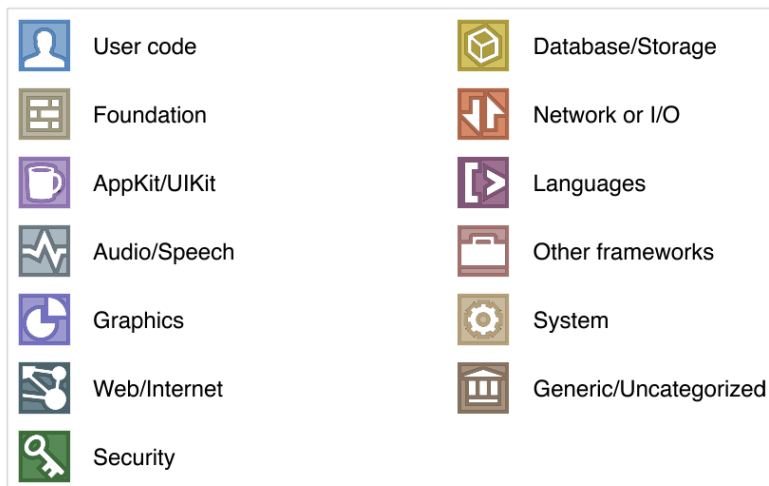


Note: When your app is paused or stops at a breakpoint, the last state of the debug gauges freezes and the detail report of each gauge at that moment can be accessed. Live indication stops.

You can hide the debug gauges to have more room for the process view display by clicking the Hide/Show debug gauges button .

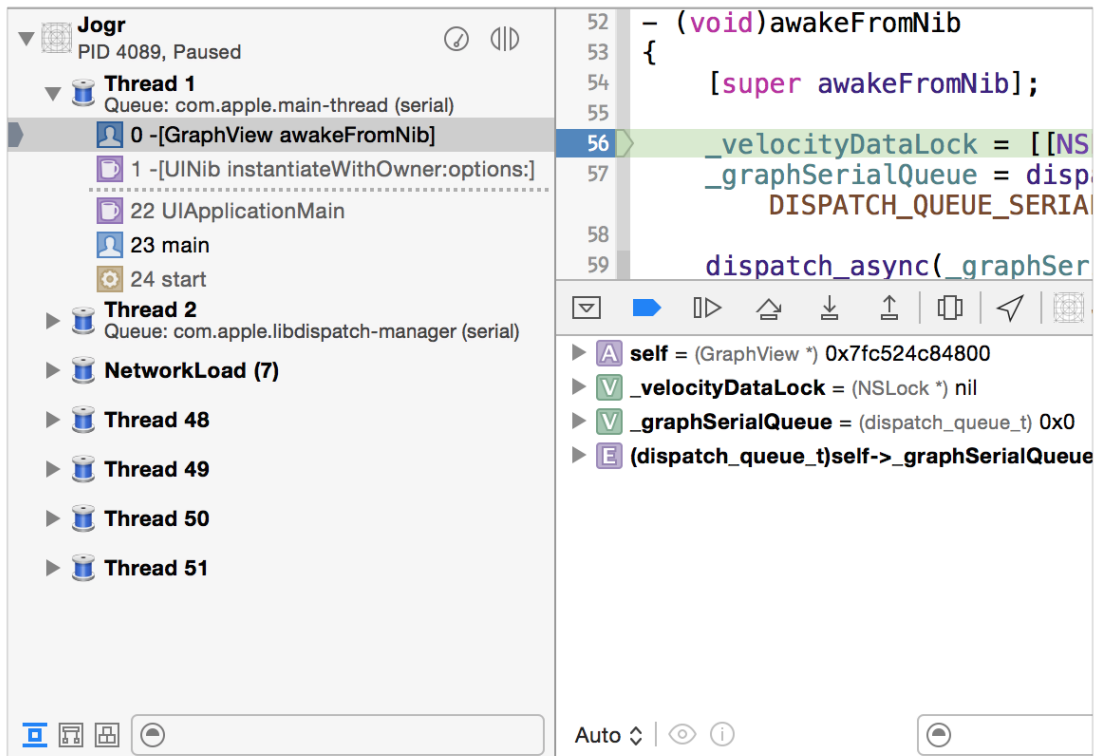
Process View Display

The process view display of the debug navigator displays the backtrace of your paused app organized by either thread or Grand Central Dispatch queues. With this tool you can debug the control flow of Swift or C-based code as well as OpenGL frames. The backtrace is displayed with each stack frame identified by an icon. The icons tell you where in the compiled code the stack frame originates.



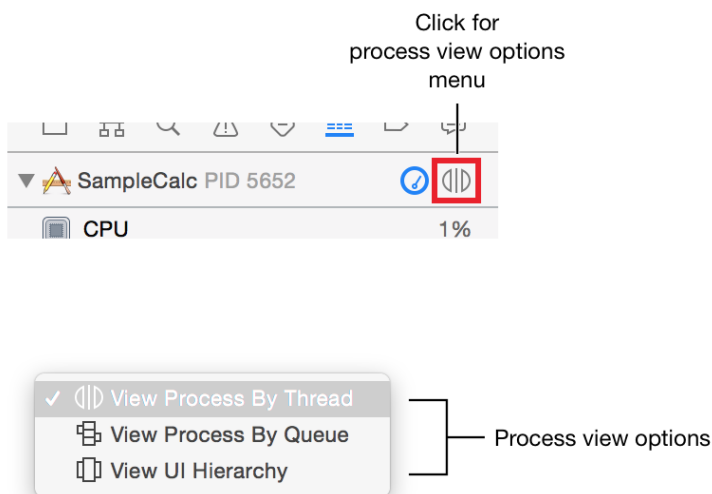
These icons are displayed in different colors to help you differentiate them easily. When they are colored gray, it means that the stack frame indicated is from the recorded backtrace and is not live in memory in the context of the paused app.

When your app pauses at a breakpoint, by default the backtrace is organized by threads—with the current stack frame highlighted and the corresponding source in the source editor showing the program counter positioned at the breakpoint. In this example, the app has paused at a breakpoint placed at line 56 in the source file in the `awakeFromNib` method. You can see the stack frame selected in the debug navigator, the source and breakpoint in the source editor, and the variable list in the debug area.



When you select another stack frame, information about the stack frame is displayed in the source editor and in the debug area. If source is not available, the source editor displays the code as decompiled assembly instructions. For a brief description of how to read the backtrace, see *Examining the Backtrace* in the *Debug Navigator* in the *Quick Start* chapter.

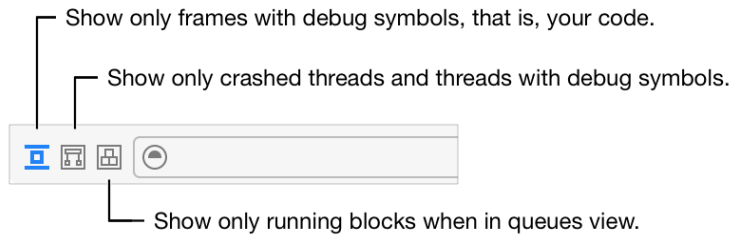
The process view options menu enables you to switch the process view display between thread organization and queues organization.



Selecting **View Process by Queue** changes the backtrace organization to show stack frames below the dispatch queue that created them. This is useful when looking for bugs that might be caused by queue interactions and blocking. See the discussion of how to use the queue display in the next chapter, *Thread and Queue Debugging*.

Note: The **View UI Hierarchy** option is also available from this menu. This option changes the debug navigator to work with bugs in the view hierarchy. It is discussed in the next chapter, see *Debugging View Hierarchies*.

The debug navigator filter bar enables you to remove extraneous information and focus on your own code during a debugging session.



- **Show only frames with debug symbols.** This filter hides threads that may not be relevant to debugging your code by suppressing the display of stack frames without debug symbols. Examples of such threads include the heartbeat and dispatch management threads and any threads that are idle and not currently executing any specific app code.
- **Show only crashed threads.** This filter displays only crashed threads and threads with debug symbols. It collapses the list to help you focus your debugging efforts by hiding calls that are far removed from your code. The presence of hidden symbols is indicated by a dotted line in the stack frame.
- **Show only running blocks.** This filter suppresses the display of non-running blocks when the debug navigator is displaying queues.

Debugging the View Hierarchy

Some bugs are immediately visible to the eye because they are problems with the views your app uses in the UI. For instance, misplaced graphics on the screen, the wrong picture associated with an item, labels and text that are incorrectly clipped or placed—these are all examples of issues with an app's view hierarchy.

Using the Xcode debugger, you can inspect the view hierarchy in detail, using a hierarchical listing of view objects, an exploded 3D rendering of your app's view hierarchy, and inspectors for object attributes and sizing. The view hierarchy tools are linked to your source code as well. These tools help you to a speedy resolution of this class of bugs.

To read a full discussion of using the Xcode debugger to debug a view hierarchy, see *Debugging View Hierarchies* in the next chapter, *Specialized Debugging Workflows*.

OpenGL ES Debugger

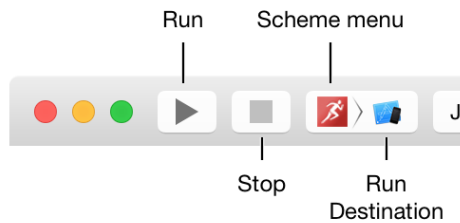
With the Xcode debugger, you have tools for debugging, analyzing, and tuning OpenGL ES and Metal apps that are useful during all stages of development. The FPS Debug Gauge and GPU report summarizes your app's GPU performance every time you run it from Xcode, allowing you to quickly spot performance issues while designing and building your renderer. Once you've found a trouble spot, you can capture a frame and use the Xcode OpenGL ES Frame Debugger interface to pinpoint rendering problems and solve performance issues.

A larger description of the features and workflow used with the OpenGL ES debugger is presented in *Debugging OpenGL ES and Metal Apps* in the next chapter, *Specialized Debugging Workflows*.

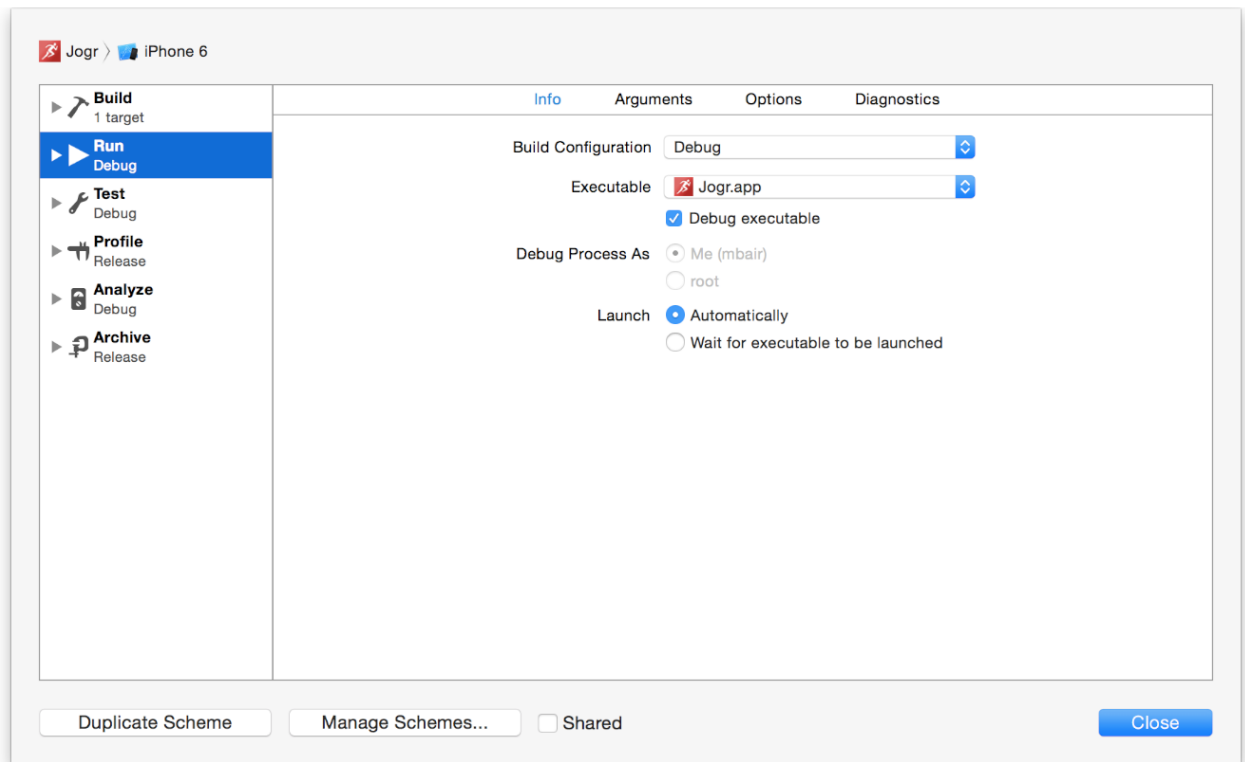
Debugging Options in the Scheme Editor

Xcode schemes control the build process. The scheme defaults created when you create a project or target are sufficient for most debugging purposes. However, useful debugging options are configurable in the scheme editor as part of the Run action configuration.

Open the scheme editor by choosing Edit Scheme from the scheme menu in the toolbar.

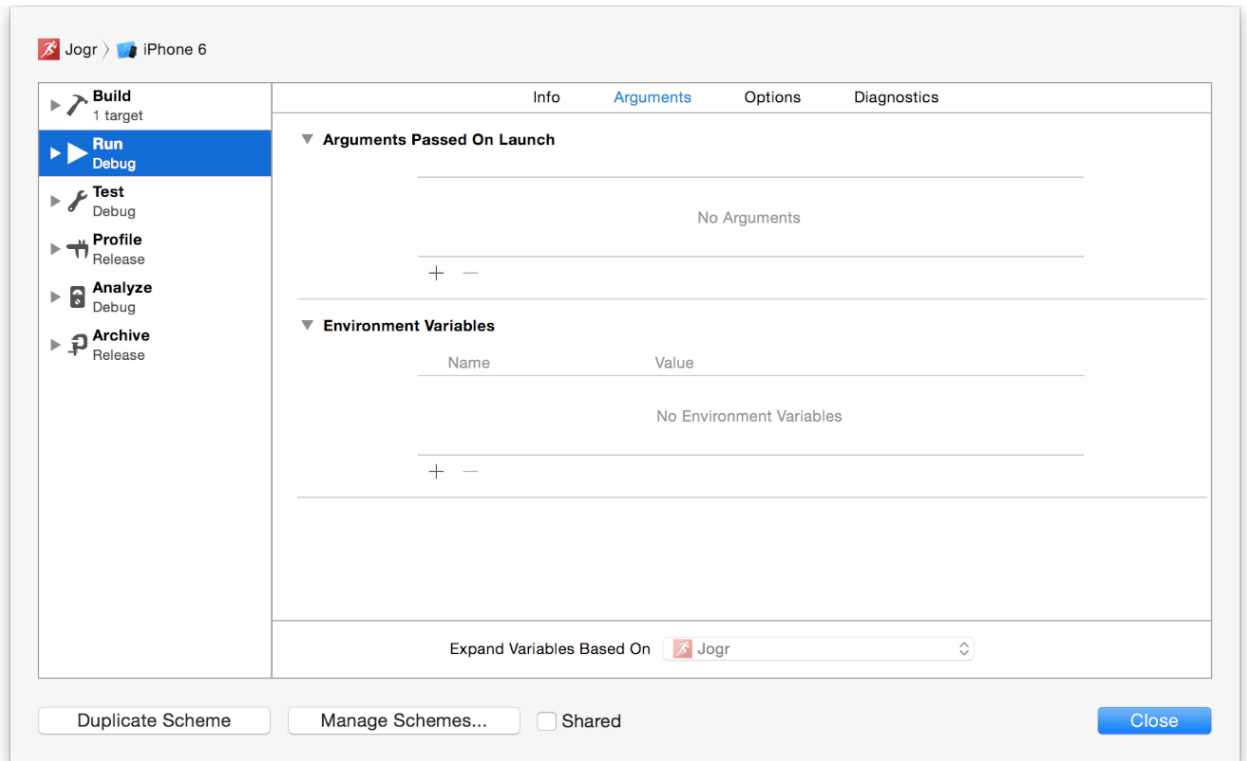


Info pane. The info pane contains settings for the build configuration.

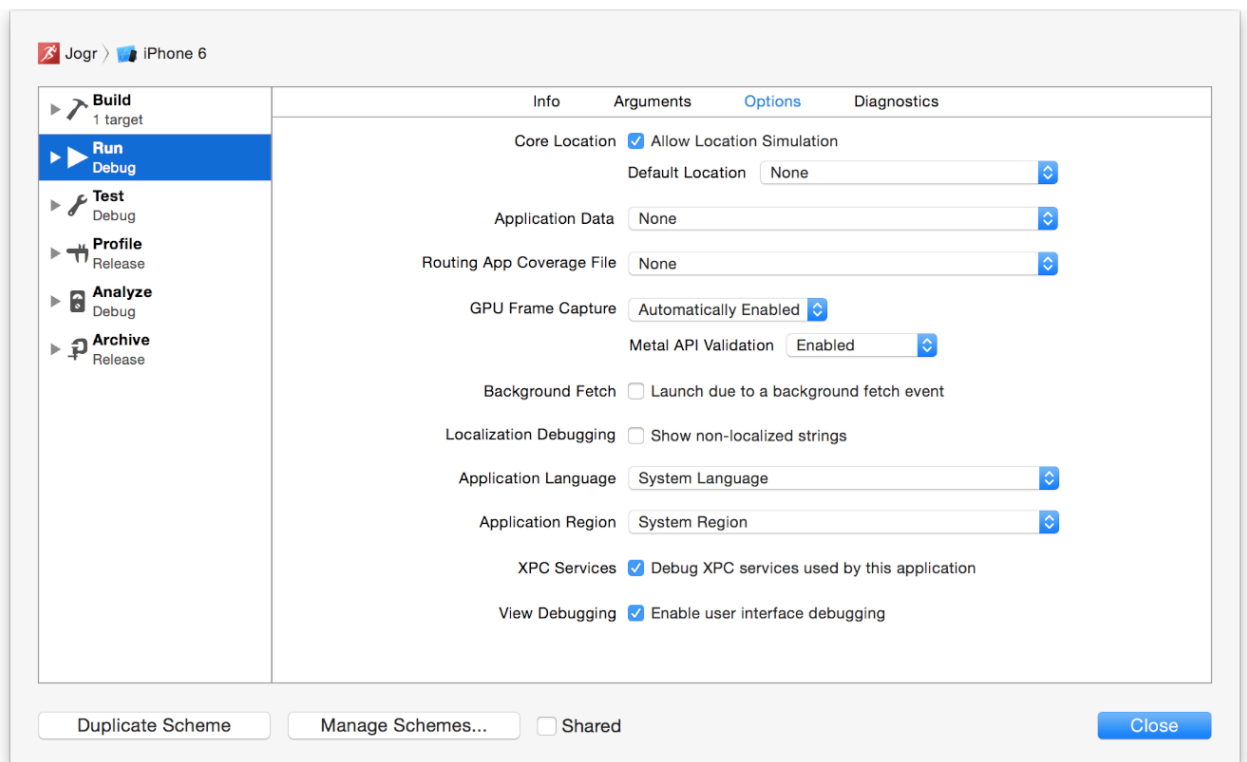


- Select the “Debug executable” checkbox if you want to run with the debugger enabled. With the debugger running, you can use Debug > Attach to Process on a process that has been launched with debugging disabled, if needed.
- Set Debug Process As to root if you are working on code that requires root privileges.

Arguments pane. With this pane, you can set up arguments to pass on launch and environment variables for your app.



Options pane. Options settings are specific to different technology needs, not just debugging.

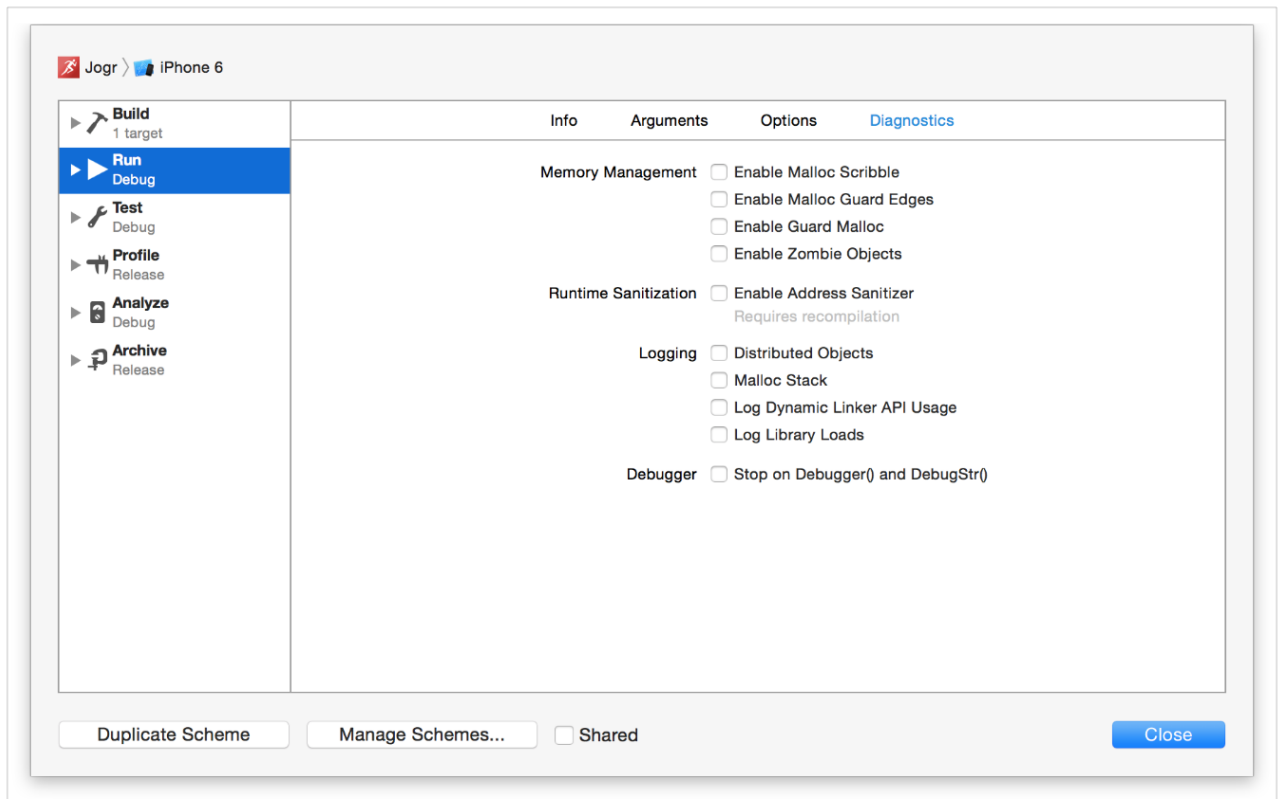


- Allow Location Simulation enables the Xcode debugger to provide location simulation services. It is on by default.
- GPU Frame Capture is enabled by default, is set to Automatically Enabled, and allows the Xcode debugger to support the OpenGL ES debugging environment. You can disable it, or limit it to either OpenGL ES or Metal technology.
- When you select “Debug XPC services used by this application,” Xcode automatically attaches to and breaks in any XPC services triggered by your app. This option makes debugging your XPC services

seamless (remember that they are separate processes). This option is on by default.

- The View Debugging option allows a special debugging library used by view hierarchy debugging to be injected into your app context when running in the debugger; it is on by default.

Diagnostics pane. The diagnostics pane allows a variety of additional diagnostic tools to be added to the Run environment.



Select the tools that you want Xcode to use. You can view output from these tools in the debug area console and in the debug log in the reports navigator.

Memory Management options:

- **Enable Malloc Scribble.** Initialize allocated memory with `0xAA` and deallocated memory with `0x55`.
- **Enable Malloc Guard Edges.** Add guard pages before and after large allocations.
- **Enable Guard Malloc.** Use `libgmalloc` to catch common memory problems such as buffer overruns and use-after-free.
- **Enable Zombie Objects.** Replace deallocated objects with a “zombie” object that traps any attempt to use it. When you send a message to a zombie object, the runtime logs an error and crashes. You can look at the backtrace to see the chain of calls that triggered the zombie detector.

Note: Using Guard Malloc and Zombie Objects diagnostics options disables the memory debug gauge.

Runtime Sanitization:

- **Enable Address Sanitizer.** Halts your app and displays the location in the debugger when memory corruption happens. Address sanitizer, working together with the Xcode debugger when it is triggered, helps you find and fix memory problems caused that are otherwise elusive to track down.

Enabling this option forces a recompilation the next time you Run your app for debugging.

Logging options:

- **Distributed Objects.** Enable logging for distributed objects (`NSConnection`, `NSInvocation`, `NSDistantObject`, and `NSConcretePortCoder`).
- **Malloc Stack.** Record stack logs for memory allocations and deallocations.

- **Log DYLD API Usage.** Log dynamic-linker API calls (for example, `dlopen`).
- **Log Library Loads.** Log dynamic-linker library loads.

Debugger options:

- **Stop on Debugger and DebugStr.** Enable Core Services routines that enter the debugger with a message. These routines send a `SIGINT` signal to the current process.

Copyright © 2016 Apple Inc. All Rights Reserved. Terms of Use | Privacy Policy | Updated: 2016-09-13