Quick Start

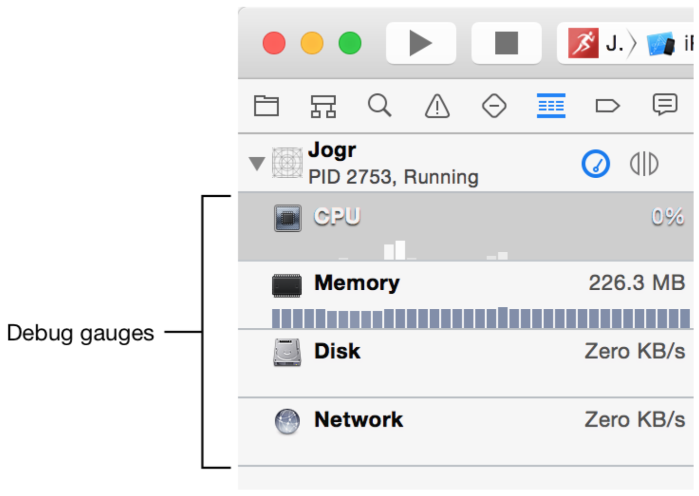
Getting Ready to Debug

Two of the most important things to do in debugging are to analyze the logic, or control flow, of the code, and to be sure the app presents correct data. Apps with logic problems exhibit unexpected behavior: Nothing happens, the wrong thing happens, or the app crashes. Code that performs incorrect operations often presents the wrong data to the user; imagine pressing buttons on a calculator to add *1* and *1* and obtaining the result *20*. You debug when you see an issue, to make sure that the app behaves as planned and that the right results are being shown to the user.

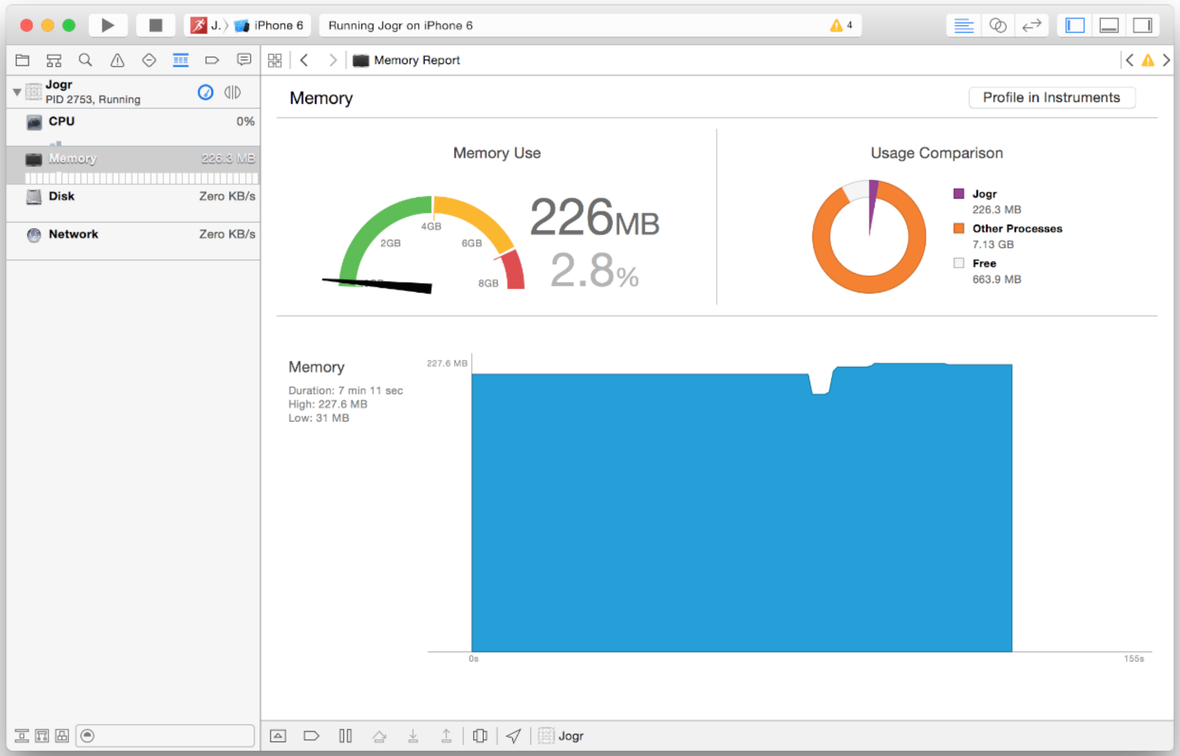
Consider that you ran your app and found that the new code you added didn't seem to be executed. You don't know yet whether the code wasn't hit or whether it did but didn't produce a visible output. But you know that something is amiss—you have a bug.

Exercising an App in the Debugger

Other problem situations can be more difficult to spot. Exercising an app in the debugger gives you the opportunity to monitor its use of system resources using the debug gauges. Pay attention to the shape of the graph, and think about what you should expect to see as the app runs and various parts of the code are called into play. For example, if you see unexpected spikes in the CPU gauge or a constantly rising pattern in the memory gauge, these might indicate subtle kinds of problems.



Clicking a gauge opens up a more detailed report for a higher-resolution view of the situation.



Whether your new code doesn’t do what you expect or you see unexpected indications in the debug gauges, you probably don’t immediately know the cause of the problem. However, you already have some information:

- The app behaved correctly before you started adding the new feature.
- The app behaved correctly up to the point where you used the app’s UI to run the new code.
- You know what new code you added to the source.

To further isolate the cause, you can set *breakpoints* in the code.

Using Breakpoints

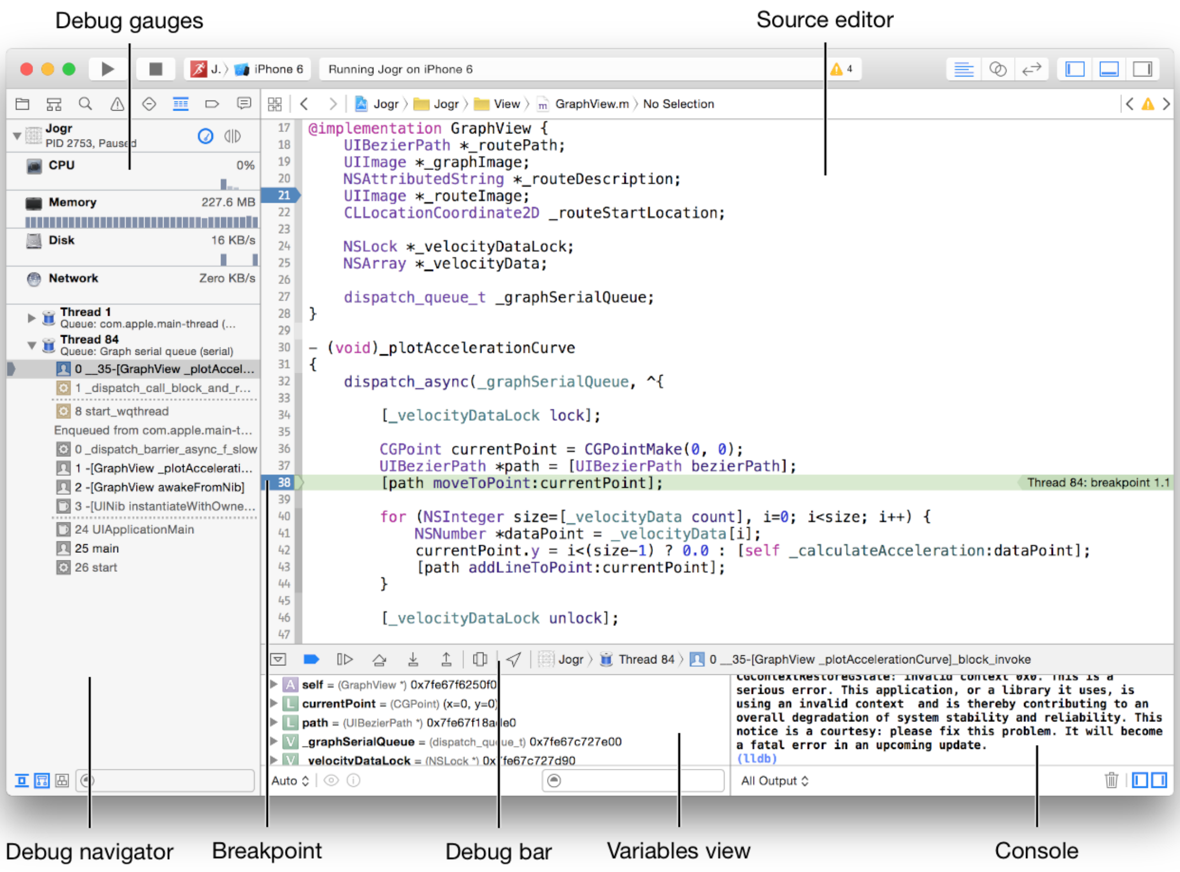
Breakpoints are a versatile solution for inspecting a running app.

A breakpoint interrupts normal execution so that you can run your app step by step with the debug bar controls to examine the flow of control and state of variables at every line of code. When your code encounters a breakpoint, the debugger pauses the app, switches to the Xcode main window with your source positioned for you to examine, and populates the variables view and the debug navigator process view with tools for looking at the state of the paused app. Breakpoints have several features you can use to modify their behavior and make it easier for you to gather information. You can add breakpoints to your source before you run the app or you can add them to the code while it is running.

To insert a breakpoint:

1. Position the source in the source editor.
2. Click the source editor gutter next to the line of source where you want the debugger to pause your app.

A file and line breakpoint is created, and it is enabled by default.

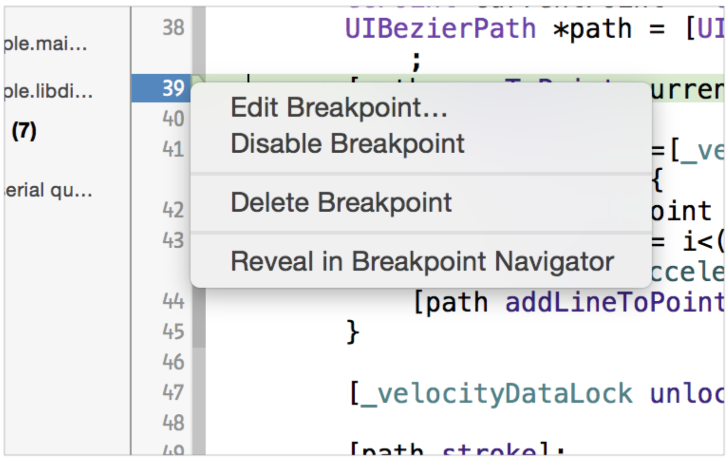

Debug gauges
Source editor
Running Jogr on iPhone 6
Jogr > Jogr > View > GraphView.m > No Selection
Jogr
PID 2753, Paused
CPU 0%
Memory 227.6 MB
Disk 16 KB/s
Network Zero KB/s
Thread 1
Queue: com.apple.main-thread (...
Thread 84
Queue: Graph serial queue (serial)
0 __35-[GraphView _plotAccel...
1 _dispatch_call_block_and_r...
8 start_wqthread
Enqueued from com.apple.main-t...
0 _dispatch_barrier_async_f_slow
1 -[GraphView _plotAccelerati...
2 -[GraphView awakeFromNib]
3 -[UINib instantiateWithOwne...
24 UIApplicationMain
25 main
26 start
Thread 84: breakpoint 1.1
Jogr > Thread 84 > 0 __35-[GraphView _plotAccelerationCurve]_block_invoke
Auto
All Output
Debug navigator
Breakpoint
Debug bar
Variables view
Console

```
@implementation GraphView {
    UIBezierPath *_routePath;
    UIImage *_graphImage;
    NSAttributedString *_routeDescription;
    UIImage *_routeImage;
    CLLocationCoordinate2D _routeStartLocation;

    NSLock *_velocityDataLock;
    NSArray *_velocityData;

    dispatch_queue_t _graphSerialQueue;
}

- (void)_plotAccelerationCurve
{
    dispatch_async(_graphSerialQueue, ^{

        [_velocityDataLock lock];

        CGPoint currentPoint = CGPointMake(0, 0);
        UIBezierPath *path = [UIBezierPath bezierPath];
        [path moveToPoint:currentPoint];

        for (NSInteger size=[_velocityData count], i=0; i<size; i++) {
            NSNumber *dataPoint = _velocityData[i];
            currentPoint.y = i<(size-1) ? 0.0 : [self _calculateAcceleration:dataPoint];
            [path addLineToPoint:currentPoint];
        }

        [_velocityDataLock unlock];
```

```
self = (GraphView *) 0x7fe67f6250f0
currentPoint = (CGPoint) (x=0, y=0)
path = (UIBezierPath *) 0x7fe67f18ade0
_graphSerialQueue = (dispatch_queue_t) 0x7fe67c727e00
velocityDataLock = (NSLock *) 0x7fe67c727d90
```

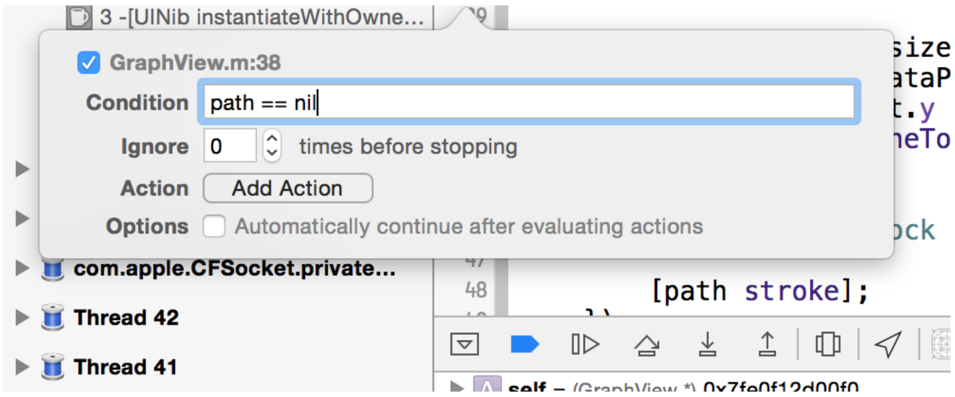
```
CGContextRestoreGState: invalid context 0x0. This is a
serious error. This application, or a library it uses, is
using an invalid context  and is thereby contributing to an
overall degradation of system stability and reliability. This
notice is a courtesy: please fix this problem. It will become
a fatal error in an upcoming update.
(lldb)
```

You use the controls in the debug bar to step in various ways, as you'll see in Controls in the Debugger. Stepping through a section of code manually is the high-resolution way to examine your code, and it can also be time consuming. Once you have put a breakpoint in your code, you can set it to operate in different ways to increase the efficiency of your debugging. Control-click the breakpoint to show the context sensitive menu, then choose Edit Breakpoint.



```
    UIBezierPath *path = [UI
    ;
            [path addLineToPoint
    }

    [_velocityDataLock unloc
```

You can configure breakpoints to halt the program based on a condition. Because your app is running when breakpoints are active, you can use any code or variable in your app that is in scope at the point of the source where the breakpoint is activated to test for a condition.

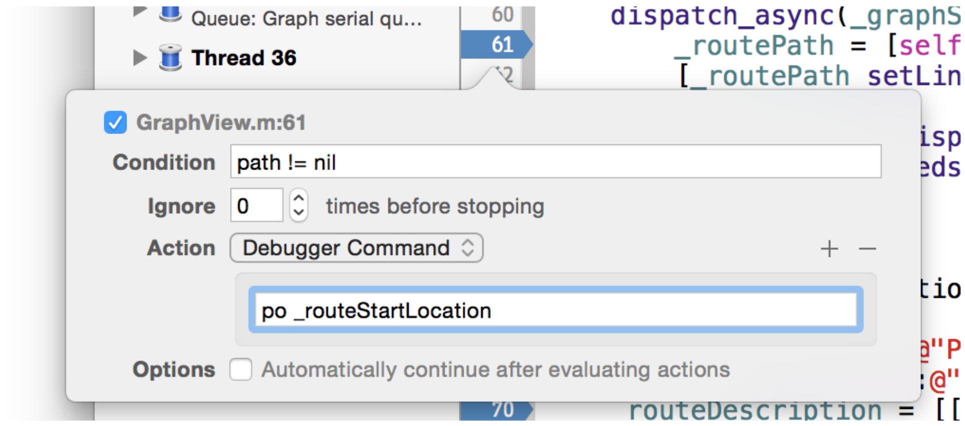


GraphView.m:38
Condition: path == nil
Ignore: 0 times before stopping
Action: Add Action
Options: Automatically continue after evaluating actions

```
[path stroke];
```

Breakpoint conditions are useful in many ways. For one example, say your new feature relies on the state of a variable. You notice that when a problem has occurred and you inspect a variable with the app paused at a breakpoint, that variable always has a particular unusual state—nil or some other unusual value. Once you know that, you can set up a breakpoint condition to monitor the variable and pause the app only when the variable has that value.

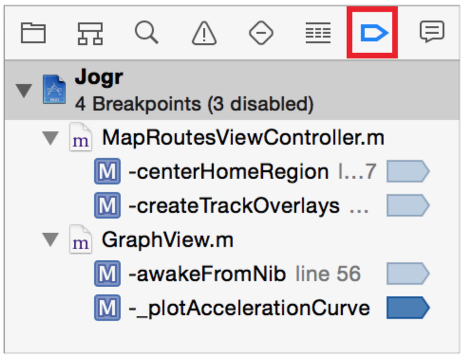
Another useful capability of breakpoints is the ability to trigger an action to be performed.



GraphView.m:61
Condition: path != nil
Ignore: 0 times before stopping
Action: Debugger Command
po _routeStartLocation
Options: Automatically continue after evaluating actions

Given the above situation, where a loop is incrementing and a variable needs to achieve a particular state, maybe you are interested in knowing what the value of the variable is each time the loop executes. You can set the breakpoint action to print the variable description to the console at each pass through the loop using the LLDB command po (po is the provided LLDB abbreviation for the “print object” command).

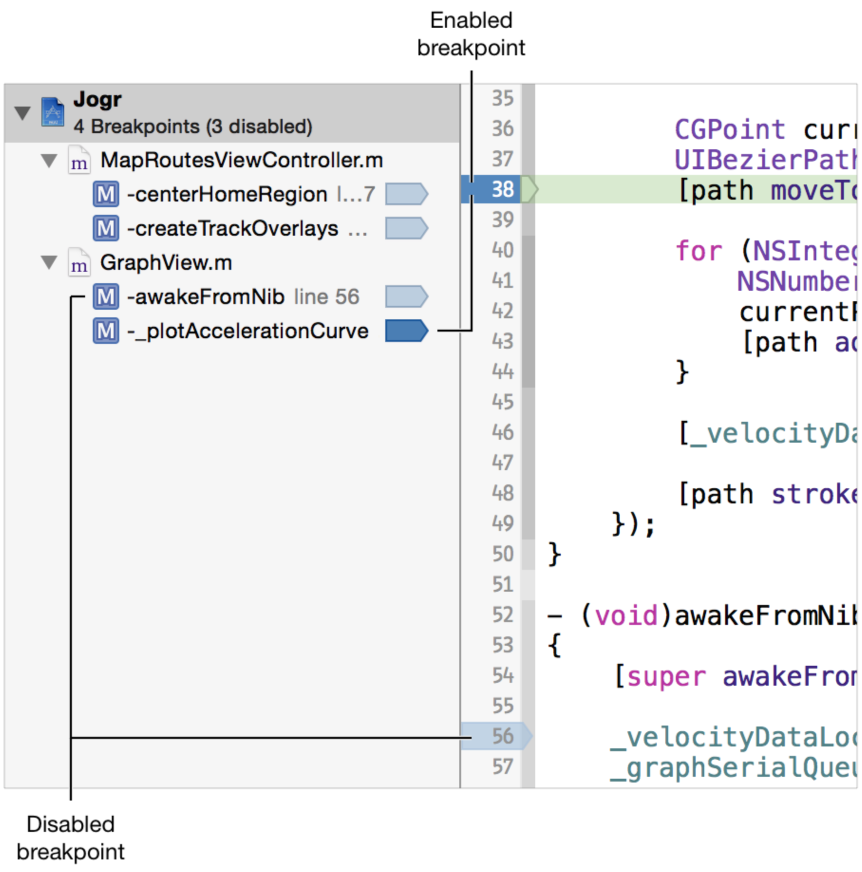
Breakpoints in your code are managed with the breakpoint navigator.



Using the breakpoint navigator, you can see all the breakpoints set in your code, edit them, enable and disable them, and change their scope of operation in the Xcode context. To enable or disable a breakpoint, click its indicator in the breakpoint navigator or in the source editor; a dimmed indicator indicates that the breakpoint is disabled.

To delete a breakpoint when you finish using it, do one of the following:

- Drag it out of the source editor gutter.
- Select it in the breakpoint navigator and press Delete.
- Control-click it (in the source editor or the breakpoint navigator) and choose Delete Breakpoint.



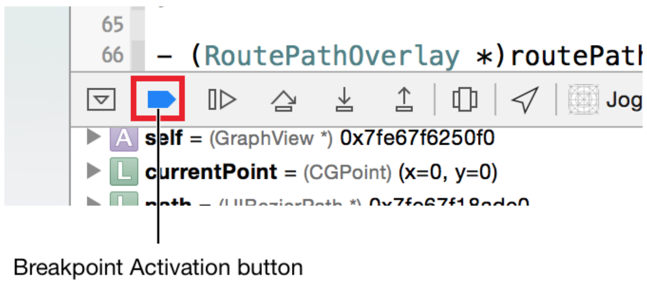
```
        CGPoint curr
        UIBezierPath
        [path moveTo

        for (NSInteg
            NSNumber
            currentP
            [path ad
        }

        [_velocityDa

        [path stroke
    });
}

- (void)awakeFromNib
{
    [super awakeFrom

    _velocityDataLo
    _graphSerialQueu
```

You can enable or disable multiple breakpoints at the same time. You might want to do that if, for example, you identify several problems located in different parts of the code.

Clicking a breakpoint row in the breakpoint navigator moves that source into the source editor at the breakpoint location.

There are times when you have placed a set of breakpoints for debugging a problem but temporarily need to run your app without having it pause so that you can reach the state at which the problem you're debugging is likely to occur. To deactivate or activate all breakpoints, click the Breakpoint Activation button in the debug bar.

Note: The Breakpoint Activation button does not change the enabled/disabled setting of the breakpoints. All breakpoints remain in place and, when activated, are enabled or disabled as they were before being deactivated.

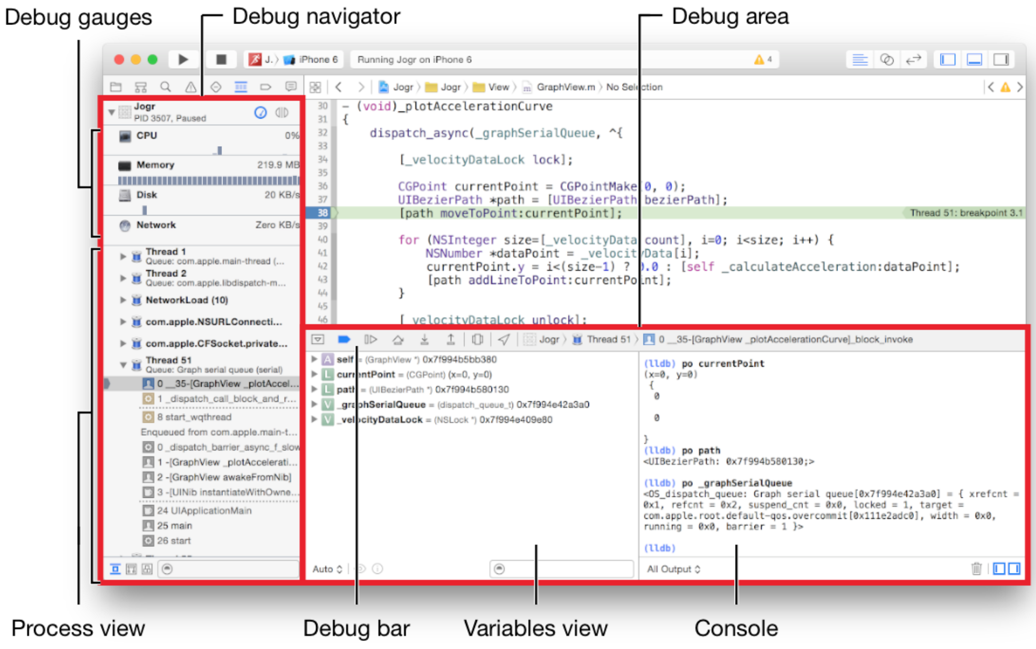


```
- (RoutePathOverlay *)routePat
```

```
self = (GraphView *) 0x7fe67f6250f0
currentPoint = (CGPoint) (x=0, y=0)
```

Controls in the Debugger

The debug navigator helps you examine the app’s control flow. When the app is paused by a breakpoint, the debug gauges display the last values generated by the app’s execution. Below the debug gauges is the process view, which can be set to show your app’s run state organized by threads or queues.


Debug gauges
Debug navigator
Debug area
Running Jogr on iPhone 6
Jogr 〉 Jogr 〉 View 〉 GraphView.m 〉 No Selection
Jogr PID 3507, Paused
CPU 0%
Memory 219.9 MB
Disk 20 KB/s
Network Zero KB/s
Thread 1 Queue: com.apple.main-thread
Thread 2 Queue: com.apple.libdispatch-m...
NetworkLoad (10)
com.apple.NSURLConnecti...
com.apple.CFSocket.private...
Thread 51 Queue: Graph serial queue (serial)
Thread 51: breakpoint 3.1
Jogr 〉 Thread 51 〉 0 __35-[GraphView _plotAccelerationCurve]_block_invoke
Auto
All Output
Process view
Debug bar
Variables view
Console

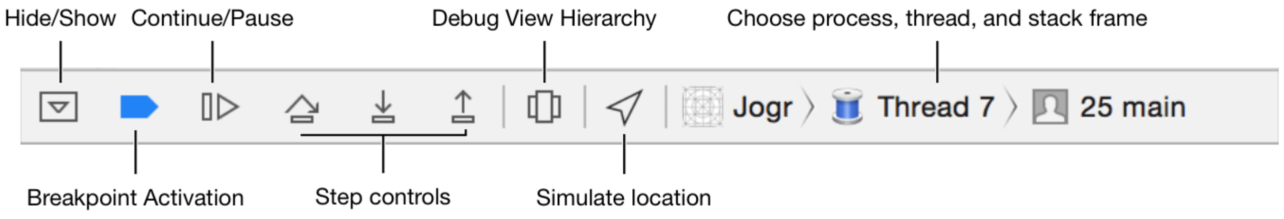
```
- (void)_plotAccelerationCurve
{
    dispatch_async(_graphSerialQueue, ^{

        [_velocityDataLock lock];

        CGPoint currentPoint = CGPointMake(0, 0);
        UIBezierPath *path = [UIBezierPath bezierPath];
        [path moveToPoint:currentPoint];

        for (NSInteger size=[_velocityData count], i=0; i<size; i++) {
            NSNumber *dataPoint = _velocityData[i];
            currentPoint.y = i<(size-1) ? 0.0 : [self _calculateAcceleration:dataPoint];
            [path addLineToPoint:currentPoint];
        }

        [_velocityDataLock unlock];
```

```
self = (GraphView *) 0x7f994b5bb380
currentPoint = (CGPoint) (x=0, y=0)
path = (UIBezierPath *) 0x7f994b580130
_graphSerialQueue = (dispatch_queue_t) 0x7f994e42a3a0
_velocityDataLock = (NSLock *) 0x7f994e409e80
```

```
(lldb) po currentPoint
(x=0, y=0)
 {
  0

  0

}
(lldb) po path
<UIBezierPath: 0x7f994b580130;>

(lldb) po _graphSerialQueue
<OS_dispatch_queue: Graph serial queue[0x7f994e42a3a0] = { xrefcnt =
0x1, refcnt = 0x2, suspend_cnt = 0x0, locked = 1, target =
com.apple.root.default-qos.overcommit[0x111e2adc0], width = 0x0,
running = 0x0, barrier = 1 }>

(lldb)
```

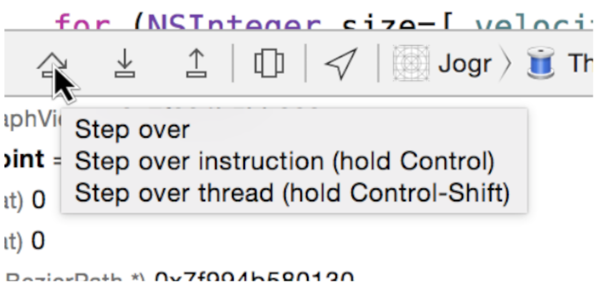
The debug area at the bottom of the Xcode main window contains three panes: the debug bar, the variables view, and the console.

Stepping Controls in the Debug Bar



The debug bar contains controls that hide and show the debug area, the Breakpoint Activation button, and the Continue/Pause button. When the app has been paused by a breakpoint, click the Continue button to resume the app.

Next to the Continue/Pause button is a set of three buttons that allow stepping your app, the step controls. Their basic operation is to step over a source instruction, step into an instruction, and step out of an instruction. If you hold the mouse over these buttons, alternative stepping modes appear, accessed by using modifier keys when clicking.


Jogr 〉 Th
Step over
Step over instruction (hold Control)
Step over thread (hold Control-Shift)

You set a breakpoint before the line of code you suspect is causing the problem. This pauses the app so that you can inspect the variables. Then you step the app to see how the variable states change, stepping over, into, and out of lines of code as needed. After you've completed the inspection, you click the Continue/Pause button to resume the app's normal operation.

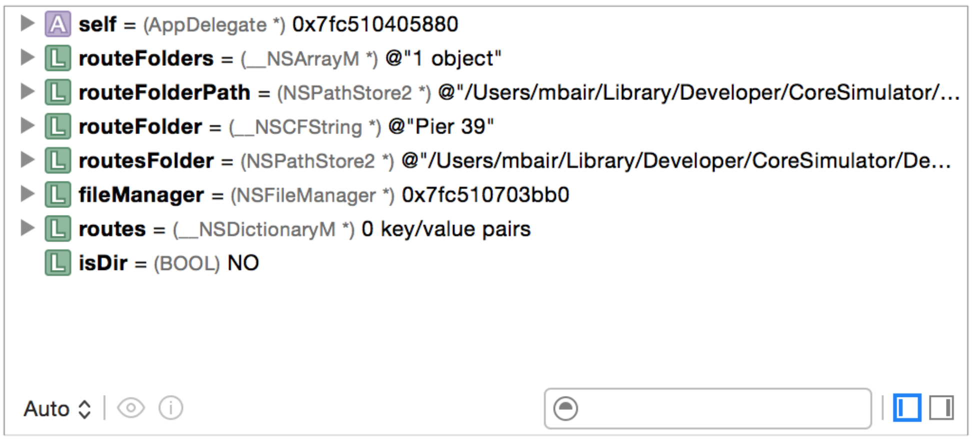
Note: The debug bar's stepping controls are also available in the Debug menu and have keyboard equivalents to make their use efficient and easy. The default key bindings can be customized using Xcode Preferences > Key Bindings.

The other buttons in the debug bar are more specialized. The button to debug view hierarchies is discussed in Debugging View Hierarchies. To the right of these two buttons, the process/thread/stack frame jump bar enables you to go directly to stack frames in the paused app, a useful alternative way to move around that works in conjunction with the process view display in the debug navigator.

When you'd like to work through a problem while seeing as much of your source surrounding the problem area as possible, you use the Hide/Show button to hide the debug area. This leaves the debug bar available for stepping; you'll see in the next section that you can use features of the debugger directly from the source editor for variable inspection when working in this manner. You reopen the debug area by pressing the Hide/Show button again.

Inspecting Variables with the Variables View

You use the variables view in the debug area to inspect variables and determine their state.

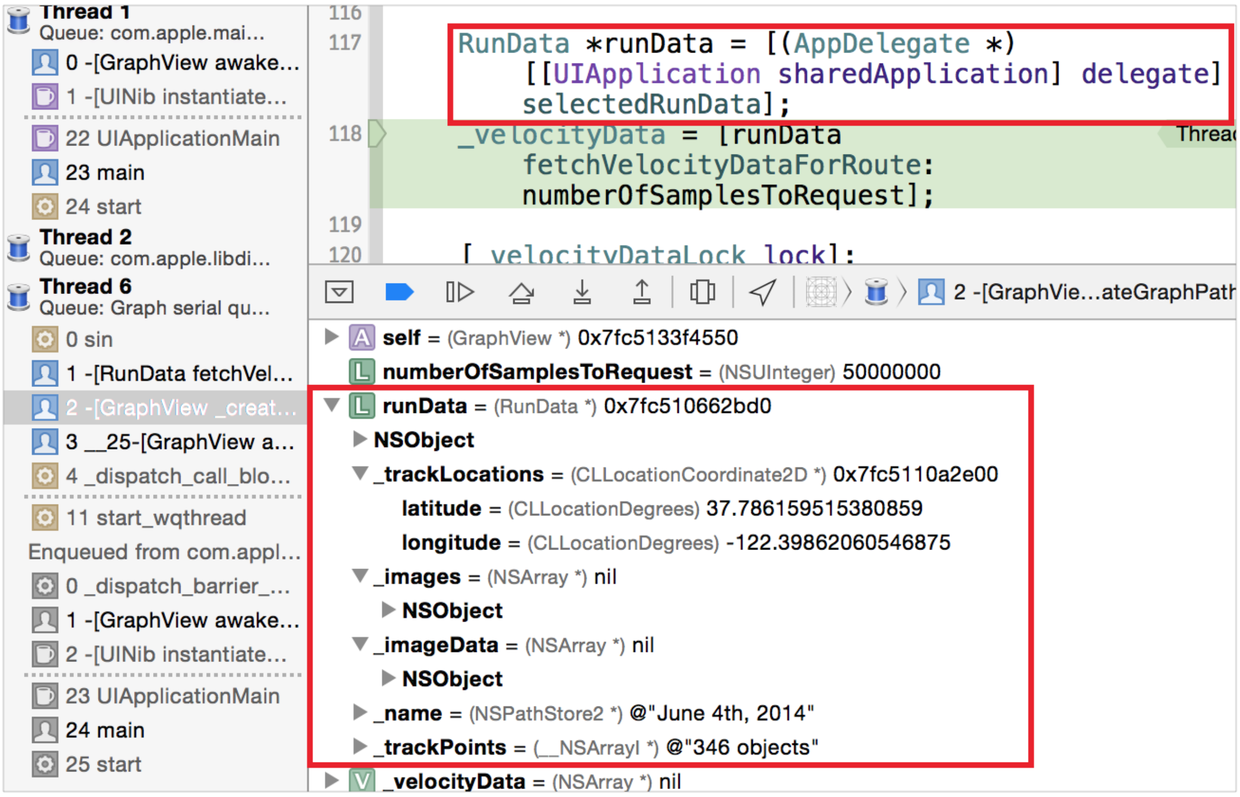


```
▶ A self = (AppDelegate *) 0x7fc510405880
▶ L routeFolders = (__NSArrayM *) @"1 object"
▶ L routeFolderPath = (NSPathStore2 *) @"/Users/mbair/Library/Developer/CoreSimulator/...
▶ L routeFolder = (__NSCFString *) @"Pier 39"
▶ L routesFolder = (NSPathStore2 *) @"/Users/mbair/Library/Developer/CoreSimulator/De...
▶ L fileManager = (NSFileManager *) 0x7fc510703bb0
▶ L routes = (__NSDictionaryM *) 0 key/value pairs
  L isDir = (BOOL) NO
```

As you can see from the figure, the variables view pane lists each variable available in the context of the breakpoint where the app is paused. Next to the disclosure triangles is a set of icons that indicate the variable's kind, followed by the variable name, followed by the variable summary representing the current value. 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.

At the lower-left corner of the variables view pane is a filter pop-up menu, set by default to Auto. This setting restricts the variables in the view to those normally considered interesting for most debugging, but you can set the variables view to show all variables (including globals, statics, and registers too) or just local variables.

For complex variable structures and objects, you can drill down into all the components of the variable by clicking the disclosure triangle. This allows you to see every detail of the variable's components at once, and let's you see how they change as you step through your app.

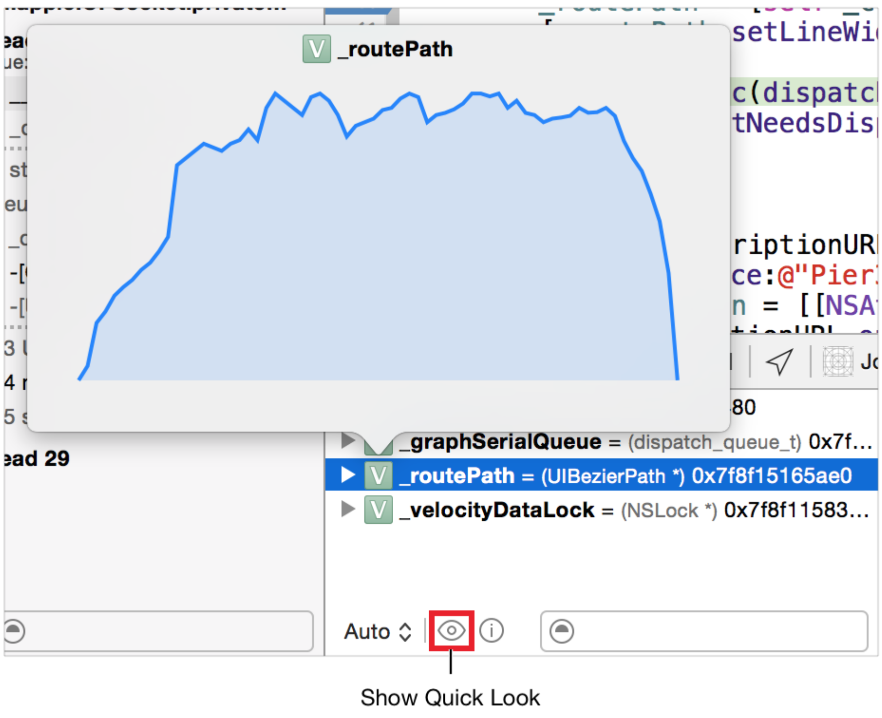


```
Thread 1
Queue: com.apple.mai...
0 -[GraphView awake...
1 -[UINib instantiate...
22 UIApplicationMain
23 main
24 start
Thread 2
Queue: com.apple.libdi...
Thread 6
Queue: Graph serial qu...
0 sin
1 -[RunData fetchVel...
2 -[GraphView _creat...
3 __25-[GraphView a...
4 _dispatch_call_blo...
11 start_wqthread
Enqueued from com.appl...
0 _dispatch_barrier_...
1 -[GraphView awake...
2 -[UINib instantiate...
23 UIApplicationMain
24 main
25 start
```

```
    RunData *runData = [(AppDelegate *)
        [[UIApplication sharedApplication] delegate]
        selectedRunData];
    _velocityData = [runData
        fetchVelocityDataForRoute:
        numberOfSamplesToRequest];

    [_velocityDataLock lock];
```

```
2 -[GraphVie...ateGraphPath
self = (GraphView *) 0x7fc5133f4550
numberOfSamplesToRequest = (NSUInteger) 50000000
runData = (RunData *) 0x7fc510662bd0
  NSObject
  _trackLocations = (CLLocationCoordinate2D *) 0x7fc5110a2e00
    latitude = (CLLocationDegrees) 37.786159515380859
    longitude = (CLLocationDegrees) -122.39862060546875
  _images = (NSArray *) nil
    NSObject
  _imageData = (NSArray *) nil
    NSObject
  _name = (NSPathStore2 *) @"June 4th, 2014"
  _trackPoints = (__NSArrayI *) @"346 objects"
_velocityData = (NSArray *) nil
```

Next to the filter pop-up menu are the Quick Look button and the Print Description button. You select a variable in the list and click these buttons to display them in popup windows. The Quick Look button produces a graphical rendering with dimension and/or value information, 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. Below is an example of the Quick Look button being used to display a `UIBezierPath` object in the variables view.



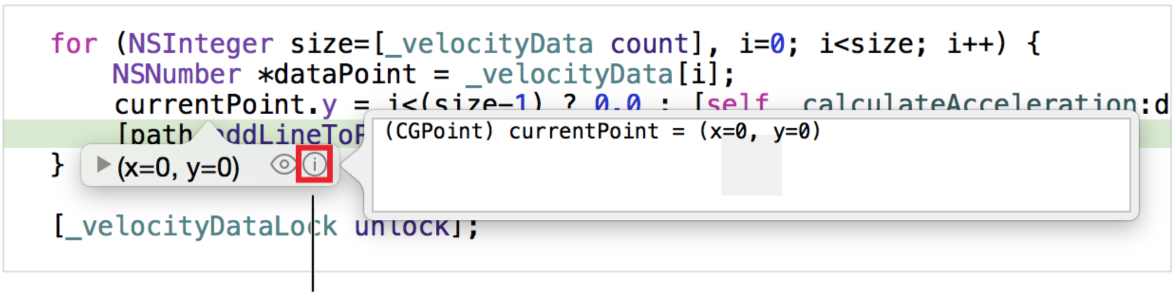
```
_graphSerialQueue = (dispatch_queue_t) 0x7f...
_routePath = (UIBezierPath *) 0x7f8f15165ae0
_velocityDataLock = (NSLock *) 0x7f8f11583...
```

Note: Quick Look displays both system provided object types and your own object types. See Quick Look Data Types for more information.

The Print Description button presents textual information about the variable, equivalent to using the `po` command in the console. When you click the Print Description button, it sends the `po` output to the console and also shows it in the popover window.

For many developers, a great way to work is to see as much of the source as possible when paused in the debugger. You hide the debug area and work directly in the source editor. Just hold the mouse over a variable in the source editor for a moment and a pop-up window appears with the same information that the variable would have displayed within the variables view. You can open the disclosure triangle, again just as in variables view, to see the components of a complex variable, and so forth. The pop-up window also has the Quick Look and Print Description buttons available right there as well: clicking them has the same effect as clicking them in variables view.

For example, this code shows a `CGPoint` variable being inspected with the Print Description button.



```
for (NSInteger size=[_velocityData count], i=0; i<size; i++) {
    NSNumber *dataPoint = _velocityData[i];
    currentPoint.y = i<(size-1) ? 0.0 : [self calculateAcceleration:d
    [path addLineToP
}
[_velocityDataLock unlock];
```

```
(x=0, y=0)
```

```
(CGPoint) currentPoint = (x=0, y=0)
```

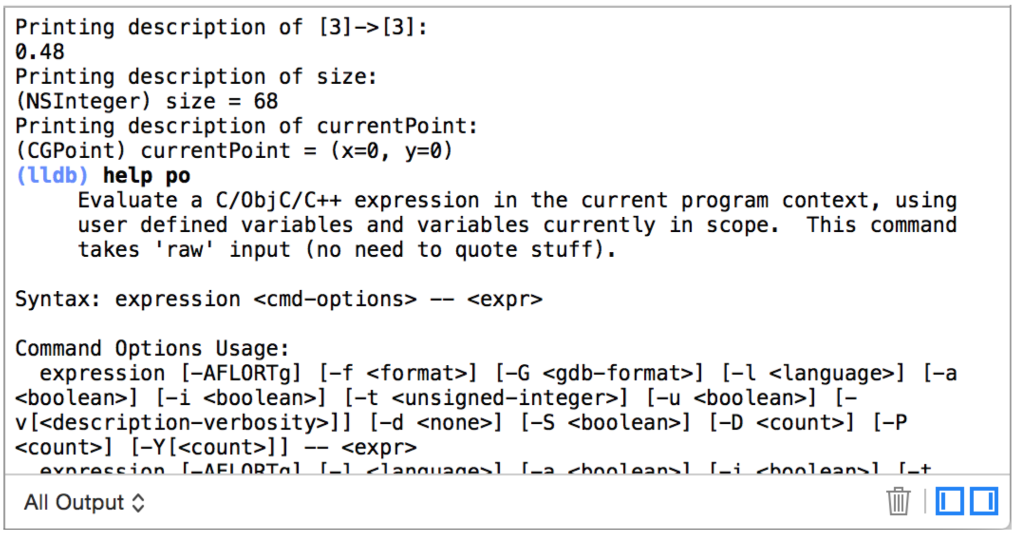
As you might expect, the Print Description button output displays both in a pop-up window and in the console.

Most of the time, you use the variables view, quick look, and print description tools to inspect the variable values when you're debugging, making sure that they are as expected. You change the code when you have a fix in mind, build and run again to test it. However, there are times when you might want to try changing a variable value on the fly to see if that corrects a problem without having to go through a full build cycle. This is easily done in the source editor using the pop-up window on a variable: Hold the mouse over a variable until the popup window appears, drill down into the variable using the disclosure triangle to the part of the variable you want, then double-click the value. Remember that changing the value of a variable dynamically is often a bit risky because you're changing the state of a running app, which might have side effects but for some kinds of debugging it can provide the information you're looking for.

Understanding the Console

The Xcode debugger area includes the console view. The console view is a Terminal-like command-line environment that records output from the app and the debugger while the debugging session is running. You use the console to collect printed variable values as you run, break, and inspect variables. The values in the console view persist throughout a debugging session, you can see a listing of outputs created there during your entire session. When you finish a session and rerun the app for another session, the console is cleared. As a result, it always starts empty at the beginning of a debugging session. However, you can review the console's contents from previous debugging sessions by going to the Report navigator and checking the contents of the debug sessions stored there.

The Xcode debugger uses LLDB, a low-level debugging engine, to perform its functions. Because the console enables you to use the LLDB command-line interface directly, you can use any LLDB commands in the console view to supplement or extend the Xcode debugging experience.



```
Printing description of [3]->[3]:
0.48
Printing description of size:
(NSInteger) size = 68
Printing description of currentPoint:
(CGPoint) currentPoint = (x=0, y=0)
(lldb) help po
     Evaluate a C/ObjC/C++ expression in the current program context, using
     user defined variables and variables currently in scope.  This command
     takes 'raw' input (no need to quote stuff).

Syntax: expression <cmd-options> -- <expr>

Command Options Usage:
  expression [-AFLORTg] [-f <format>] [-G <gdb-format>] [-l <language>] [-a
<boolean>] [-i <boolean>] [-t <unsigned-integer>] [-u <boolean>] [-
v[<description-verbosity>]] [-d <none>] [-S <boolean>] [-D <count>] [-P
<count>] [-Y[<count>]] -- <expr>
```

LLDB is a powerful debugging environment with many capabilities. Learn the basics of using LLDB by reading *LLDB Quick Start Guide*.

Examining the Backtrace in the Debug Navigator

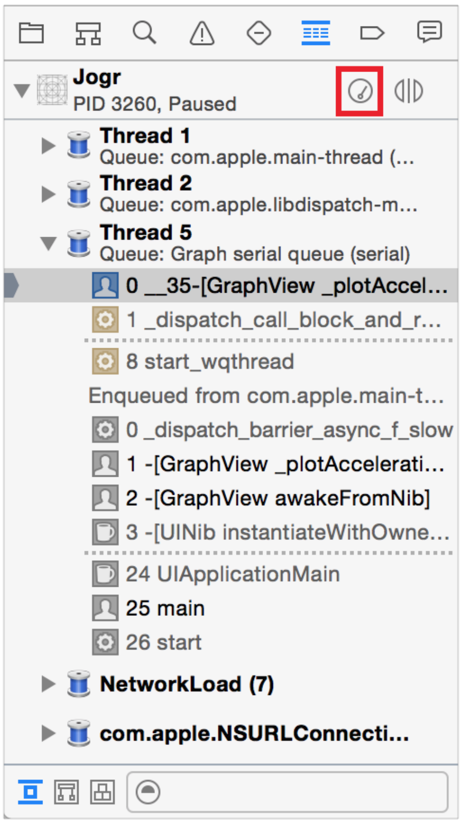
To recap the situation described earlier in this chapter, you launched your app using the Run button. You tried to invoke the new code that you added and nothing happened. Now you've added a breakpoint where the new code should be started from and tried to invoke the code again. The Xcode debugger has paused your app at that location and populated the variables view in the debug area and the debug navigator with the current state of the app.

You'll likely first want to take a look at the variables and their values. The variables view, Quick Look, and Print Description tools give you that first look into the state of the app. The next question that needs to be addressed is "How did the code get here?" To answer this question, you use the debug navigator process view pane.

When the app stops at a breakpoint, the debugger "unwinds" the program flow and presents that to you in the debug navigator process view pane. Some definitions will make this clear:

- A *stack frame* is an instance of an invocation of a method.
- *Unwinding* means to "look at the method that contains your breakpoint and follow the stack frame pointers back to the call site." The debugger unwinds until it reaches the beginning of the thread.
- The sequence of stack frames unwound in this fashion is known as the *backtrace*.

By default, this view is organized by threads with the current stack frame highlighted, corresponding to the source in the source editor where the program counter is positioned at the breakpoint. (The debug gauges have been hidden in this example of the debug navigator using the highlighted button, allowing you to concentrate on the process view showing the backtrace.)


Jogr
PID 3260, Paused
Thread 1
Queue: com.apple.main-thread (...
Thread 2
Queue: com.apple.libdispatch-m...
Thread 5
Queue: Graph serial queue (serial)
0 __35-[GraphView _plotAccel...
1 _dispatch_call_block_and_r...
8 start_wqthread
Enqueued from com.apple.main-t...
0 _dispatch_barrier_async_f_slow
1 -[GraphView _plotAccelerati...
2 -[GraphView awakeFromNib]
3 -[UINib instantiateWithOwne...
24 UIApplicationMain
25 main
26 start
NetworkLoad (7)
com.apple.NSURLConnecti...

The backtrace allows you to understand the control flow in your app. It is displayed in the debug navigator's process view pane with each stack frame identified by an icon. The icons tell you where in the compiled code the stack frame is from. There are icons for user code, the Foundation framework, AppKit or UIKit frameworks, various graphics frameworks, and so forth. The debugger retrieves and splices-in this full history of execution, even including stack frames that are no longer in memory.

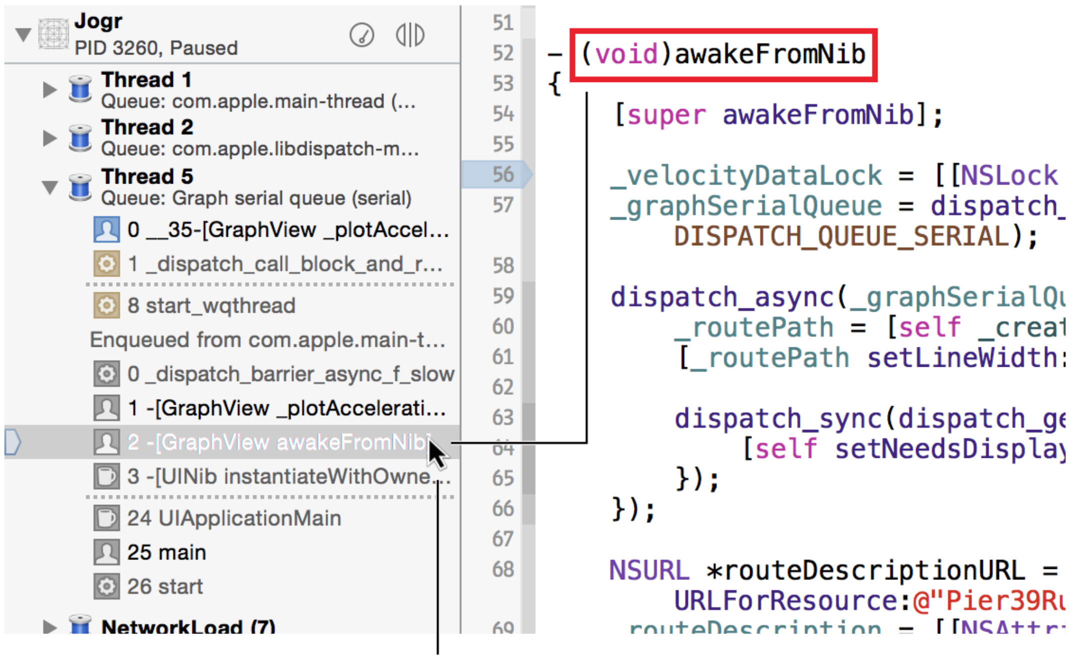
With this view of the backtrace, you can see how your app came to its current state at the breakpoint pause. Looking at the illustration above:

1. The app is paused in your source code at `_plotAccel`, which is executing in thread 5.
2. This code has been called by a system function that is part of a dispatched asynchronous callback.
3. Some system stack frames are elided from the view—the dotted line represents them. You can show these stack frames by deselecting the leftmost button in the filter bar at the bottom of the debug navigator.
4. The stack frame below the dotted line shows the start of the worker thread that called the code.
5. Below that, you can see where the block was asynchronously enqueued in the recorded backtrace.

   The recorded backtrace shows that this block was enqueued when `awakeFromNib` was running, possibly near the start of the app.

As you click on the stack frames in the backtrace, the source editor jumps to that point in the source or in the decompiled binary executable. For stack frames of your code, you can see your source code. If the frame is in memory, you can see variable values; if the stack frame is no longer in memory—that is, it's a recorded stack frame—the variable values are no longer available.

Note: In the debug navigator process view, if the icon next to a stack frame is colored, the stack frame is in memory. Gray icons next to a stack frame indicate that the stack frame is no longer in memory and that the debug navigator has retrieved the stack frame by analyzing the recorded call history; the debugger cannot re-create the variable values from the recorded call history.



Jogr
PID 3260, Paused
- Thread 1
  Queue: com.apple.main-thread (...
- Thread 2
  Queue: com.apple.libdispatch-m...
- Thread 5
  Queue: Graph serial queue (serial)
  - 0 __35-[GraphView _plotAccel...
  - 1 _dispatch_call_block_and_r...
  - 8 start_wqthread
  - Enqueued from com.apple.main-t...
  - 0 _dispatch_barrier_async_f_slow
  - 1 -[GraphView _plotAccelerati...
  - 2 -[GraphView awakeFromNib]
  - 3 -[UINib instantiateWithOwne...
  - 24 UIApplicationMain
  - 25 main
  - 26 start
- NetworkLoad (7)

```
- (void)awakeFromNib
{
    [super awakeFromNib];

    _velocityDataLock = [[NSLock
    _graphSerialQueue = dispatch_
        DISPATCH_QUEUE_SERIAL);

    dispatch_async(_graphSerialQu
        _routePath = [self _creat
        [_routePath setLineWidth:

        dispatch_sync(dispatch_ge
            [self setNeedsDispla
        });
    });

    NSURL *routeDescriptionURL =
        URLForResource:@"Pier39Ru
    routeDescription = [[NSAttr
```

Using the debug navigator and the backtrace, you can see whether the app's code has moved along the expected path to the breakpoint or whether it diverged from where you expected it to be. By jumping to user code in the backtrace when you have a problem, you can set new breakpoints that stop the app at earlier points and see where it went awry by examining the variables along the way.

If your new code doesn't run or display because a conditional was not fulfilled correctly, find the right point in the backtrace, set a breakpoint that will illuminate the problem, click Run/Pause to resume normal operation, and then perform the action that should run the new code again. This time, stop at the earlier point in the program flow and inspect variables, and then step through the app until you find the cause of the problem, examining the variables along the way.

Cycling Through the Debugging Process

Debugging is an iterative effort. You discover a problem, you set breakpoints to help you locate where in the source it is happening, you inspect the backtrace and variables to assess their state and the cause of the problem, and you devise a solution or fix. With a potential solution in hand, you make a change to the source and rerun the app to see whether the problem is solved. If testing shows that the problem persists, you repeat this cycle until you've put the right fix in place.

When a problem is solved, you can disable or delete the breakpoints you used in its investigation. This is because you don't want them to get in the way as you move on to the next problem.