Instruments User Guide

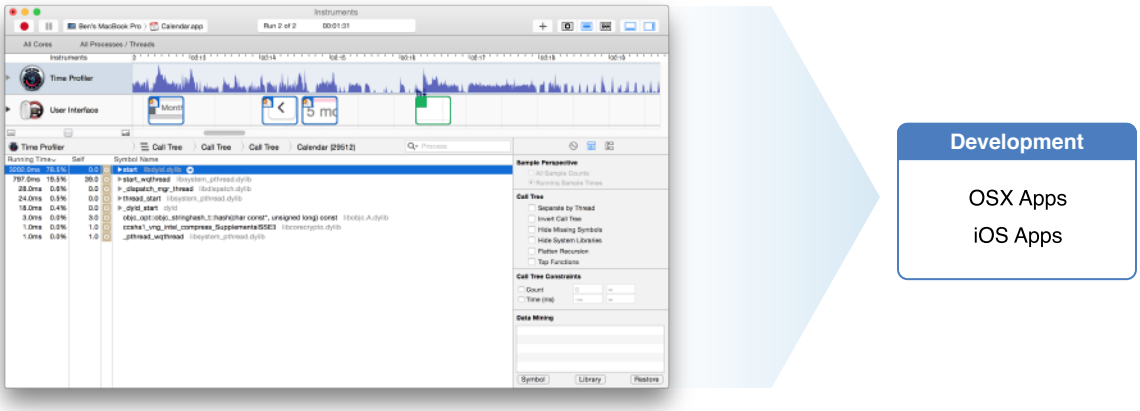
Developer

Contents

Figures, Tables, and Listings

About Instruments

Instruments is a performance-analysis and testing tool for dynamically tracing and profiling OS X and iOS code. It is a flexible and powerful tool that lets you track a process, collect data, and examine the collected data. In this way, Instruments helps you understand the behavior of both user apps and the operating system.



At a Glance

With Instruments, you use special tools (known as *instruments*) to trace different aspects of a process's behavior. You can also use the tool to record a sequence of user interface actions in OS X and replay them over and over again to replicate a set of steps, while using one or more instruments to gather data.

Instruments provides the ability to:

- Examine the behavior of one or more processes
- Record a sequence of OS X user actions and replay them, reliably reproducing those events and collecting data over multiple runs
- Profile apps in OS X and in iOS (in iPhone or iPad Simulator, or on a physical iOS device)
- Create your own custom DTrace instruments to analyze aspects of system and app behavior
- Save user interface recordings and instrument configurations as templates

Using Instruments, you can perform tasks such as:

- Track down difficult-to-reproduce problems in your code
- Do performance analysis on your app
- Find memory leaks and other problems in your app
- Automate testing of your app
- Stress-test parts of your app
- Perform general system-level troubleshooting
- Gain a deeper understanding of how your app works

Instruments is available with Xcode 3.0 and later and with OS X v10.5 and later.

Organization of This Document

The following chapters describe how to use Instruments:

- Instruments Quick Start (page 12) explains how to install Instruments and provides a quick example that walks you through gathering data.
- Touring Instruments (page 19) gives a brief overview of the Instruments app and introduces the main window.
- Adding and Configuring Instruments (page 22) describes how to add and configure instruments and run them to collect data on one or more processes.
- Collecting Data on Your App (page 35) describes the ways you can initiate traces and gather trace data.
- Examining Your Collected Data (page 45) describes the tools you use to view the data returned by the instruments.
- Saving and Importing Trace Data (page 55) describes how you save trace documents and data and how you import data from other sources.
- Locating Memory Issues in Your App (page 61) provides examples of how to use the memory-oriented trace templates.
- Measuring I/O Activity in iOS Devices (page 75) provides examples of how to use the I/O-oriented trace templates.
- Measuring Graphics Performance in Your iOS Device (page 79) provides examples of how to use the OpenGL ES–oriented trace templates.
- Analyzing CPU Usage in Your App (page 85) provides examples of how to use the CPU-oriented trace templates.
- Automating UI Testing (page 93) provides examples of how to write scripts for automatic testing of your app.

- Creating Custom Instruments (page 121) shows how to create and configure your own DTrace-based custom instrument.
- Preferences (page 136) provides an overview of the preferences available in Instruments.
- Keyboard Shortcuts (page 144) provides an overview of the keyboard shortcuts available in Instruments.
- Menu Bar Definitions (page 148) provides an overview of the menus in Instruments.
- Trace Template Contents (page 154) provides an overview of the standard trace templates and the instruments they contain.

See Also

Instruments is best used in conjunction with Xcode. For information on how to use Xcode, see *Xcode Overview*. For a complete list of help articles for performing essential Instruments tasks, see *Instruments Help*.

Instruments Quick Start

Instruments is a powerful tool you can use to collect data about the performance and behavior of a process on the system, and track that data over time. Unlike most other performance and debugging tools, Instruments lets you gather widely disparate types of data and view them side by side. In this way, you can identify trends that might otherwise be hard to spot with other tools. For example, your app may exhibit large memory growth that is caused by multiple network connections being opened. Using the Allocations and Connections instruments together, you can identify the connections that are not closing and resulting in the rapid memory growth.

The Instruments tool uses individual data collection modules known as *instruments* to collect data about a process over time. Each instrument collects and displays a different type of information, such as file access, memory use, and so forth. Instruments includes a library of standard instruments, which you can configure and use to examine various aspects of your code. You can also build your own custom instruments that use DTrace to gather other kinds of data.

> **Note:** Several Apple apps—namely, iTunes, DVD Player, and apps that use QuickTime—prevent the collection of data through DTrace in order to protect sensitive or copyrighted data.

All work in Instruments is done in a *trace document*, which contains a set of instruments and the data they collected. You can save a trace document to preserve the trace data you have gathered and open it again later to view that data.

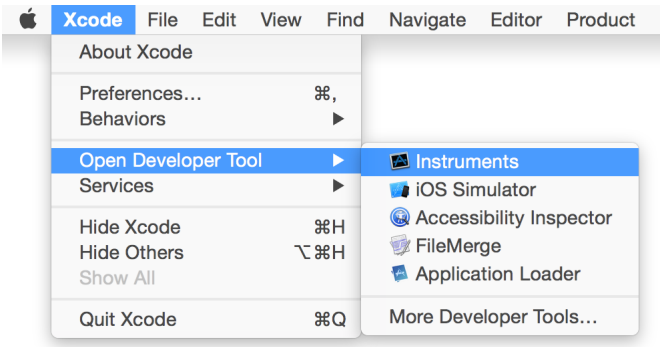
Launching Instruments

Instruments is contained within the Xcode toolset, which can be installed through the App Store. After you have installed Xcode, Instruments can be launched in one of three ways.

To run Instruments from Xcode

1. Open Xcode.

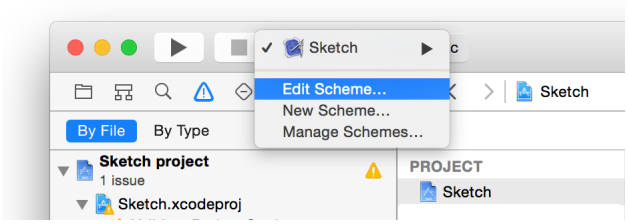
2. Choose Xcode > Open Developer Tool > Instruments.



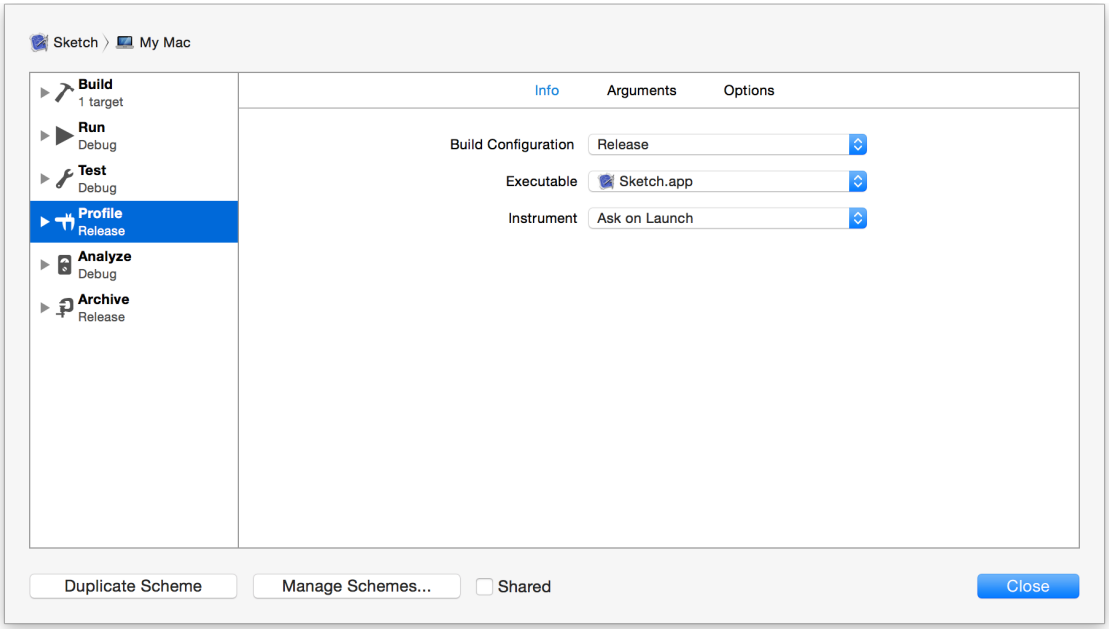
Incorporating Instruments into your workflow from the beginning of the app development process can save you time later by helping you find issues early in the development cycle. The ability to initiate Instruments directly from within Xcode allows you to gather information about your app regularly, such as every time you build it.

To run Instruments while building your code

1. Select Edit Scheme in the scheme menu in the Xcode toolbar.



2. Click Profile in the scheme editor sidebar.



3. Choose an instrument template from the Instrument pop-up menu.

   The default choice is “Ask on Launch,” which causes Instruments to display its template chooser dialog when it starts up. You can, of course, select a specific instrument profiling template here instead, if you prefer.

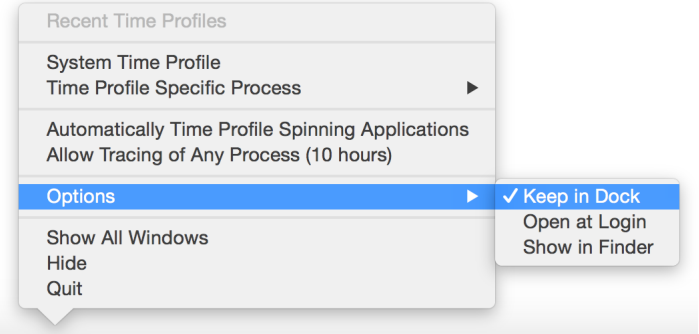
4. Click Close.
5. Hold down the Run button in the Xcode toolbar and choose Profile or choose Profile from the Product menu whenever you want to build your app and run it in Instruments.

The third way to open Instruments is from the Dock. To do this, you need to first open Instruments using one of the methods listed above and configure it to stay in the Dock.

To run Instruments from the Dock

1. Launch Instruments from Xcode using one of the two methods above.
2. Control-click the Instruments icon in the Dock.

3. Click Options > Keep in Dock.



Gathering Your First Data

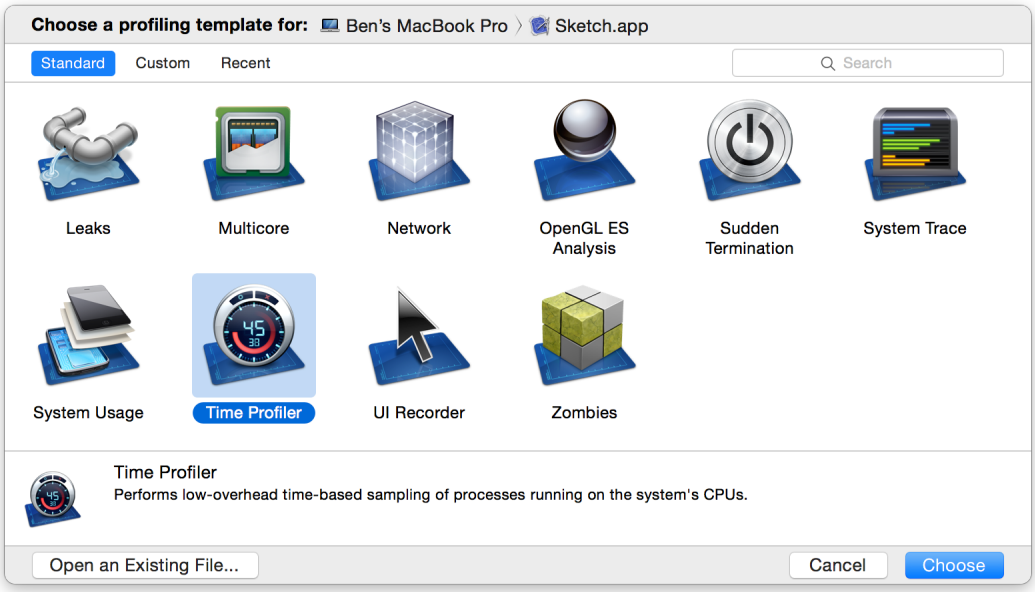
Even though each instrument is different, there is one general workflow when collecting information from your app. You:

1. Open Instruments.
2. Choose a target device and app.
3. Choose a trace template.
4. Collect information from your app.
5. Examine the collected information.

Choose a Trace Template

When Instruments first starts up, it provides you with a list of trace templates. These templates contain collections of instruments that are commonly used together to provide you with a specific set of information. Optionally, you may specify a target device and app at the top of the template selection window. In Figure 1-1, the Time Profiler trace template is selected and an app named Sketch is selected as the target app.

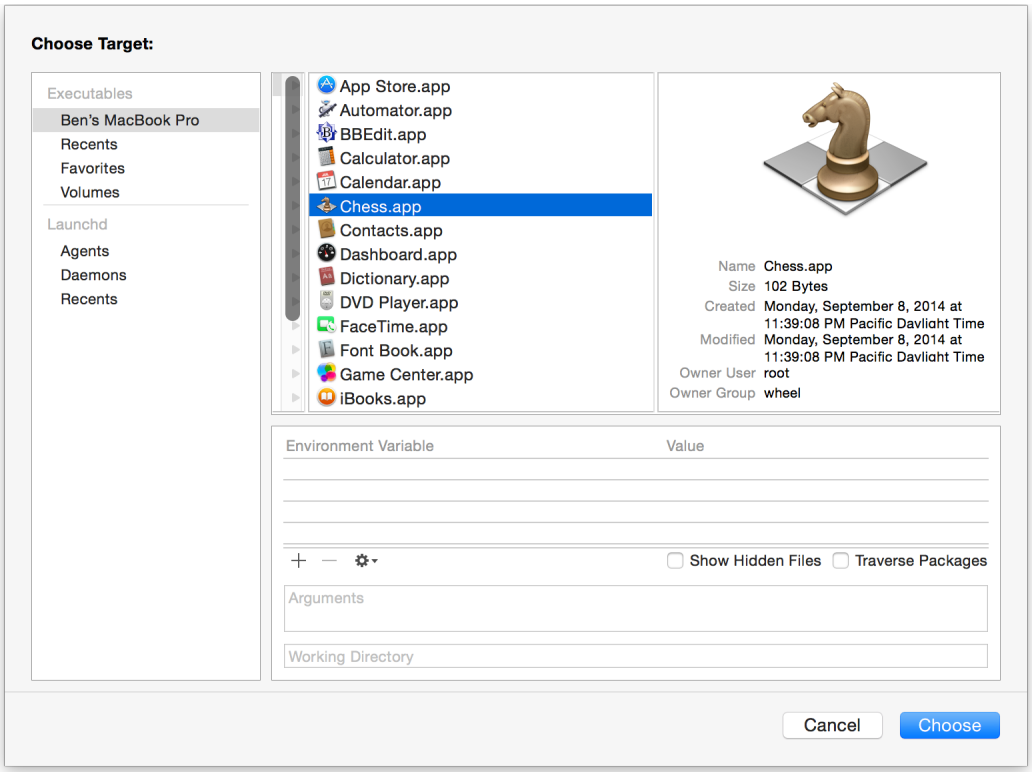
Figure 1-1 The Instruments trace template selection window



Choose Your Target

If you didn't launch Instruments from the Xcode Product menu or Run button, or select a target from the template selection window, then you need to specify a target for the instruments in the trace document to profile. A selected target may be an installed app, such as Chess in Figure 1-2, or a process that is already running.

Figure 1-2 Selecting the Chess app as the target



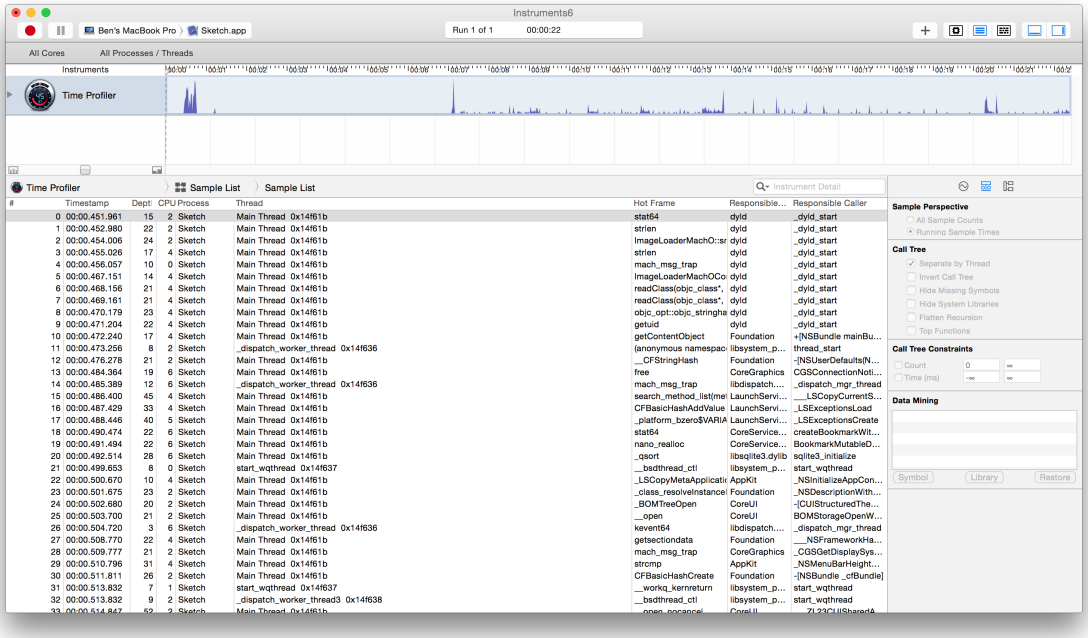
Tip: Some instruments, such as Activity Monitor and Time Profiler, allow you to gather information from all of the running processes on your device.

Collect and Examine the Data

Once you have created a trace template and chosen a target for your app, it's time to collect some data. You can adjust the information that's collected by an instrument via the Record Settings area in the Inspector sidebar. Click Record to start collecting data. When you have enough, click Stop. Figure 1-3 shows data collected from the Sketch app using the Time Profiler template.

After collecting your data, you can examine it in the detail pane, as well as the Extended Detail area in the Inspector sidebar. Select a row in the detail pane to bring up extra information about the row in the Extended Detail area.

Figure 1-3 Collecting information with the Time Profiler trace template



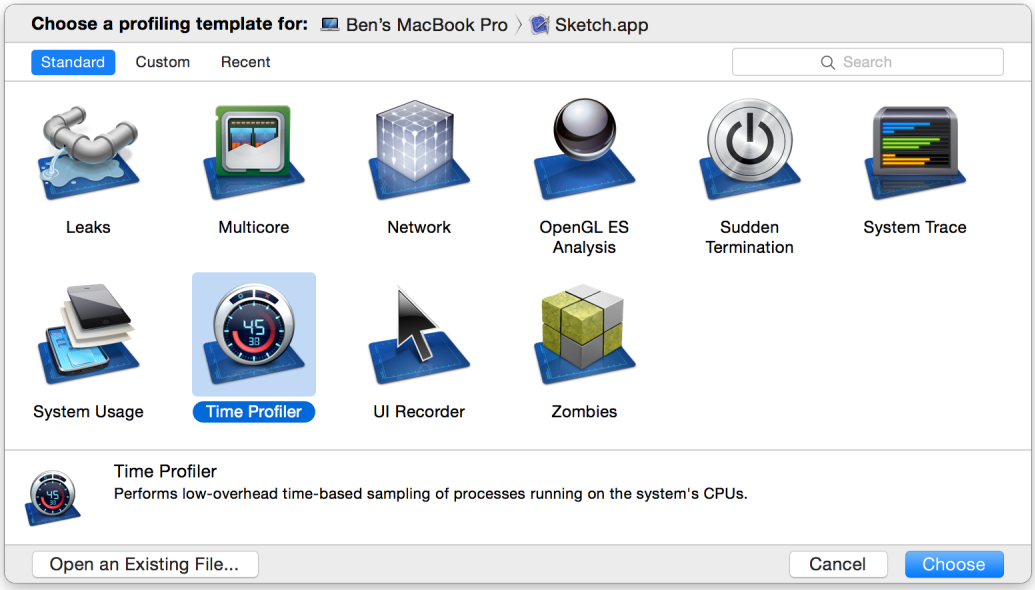
Touring Instruments

Instruments provides you with numerous ways to collect and examine information about your app. Instruments consists of two main windows — the template selection window and the trace document window. The template selection window appears when Instruments is first started and allows you to select a trace template (a group of preconfigured instruments) based on their function, as well as an app to profile. After you select a template, the trace document window containing the template's instruments and their configurations is displayed. You can add more instruments to the trace window, adjust instrument settings, collect data, examine collected data, and much more. This chapter describes these two primary windows and provides a quick walkthrough for each one.

Your First View: The Trace Template Selection Window

When Instruments opens, you are presented with a list of preconfigured trace templates, as seen in Figure 2-1. It includes a set of standard templates based on commonly performed tasks, as well as any custom templates you may have created. You can also bring up the template selection window by choosing File > New when running Instruments.

Figure 2-1 The Instruments trace template selection window



The trace template chooser dialog consists of these primary areas:

- A target chooser, for selecting the device and process(es) to profile.
- A filter for selecting sets of standard, custom, or recently used templates.
- A search field, for searching the titles and descriptions of templates.
- The (possibly filtered) list of trace templates.
- A short description of the selected trace template. Use this to determine if the chosen template is the correct template for your needs.

Use a trace template as a starting point to collect data for a particular purpose. After choosing the template to load, click Choose to open the trace document and populate the document with the template's instruments. You can then add or remove individual instruments from the trace document.

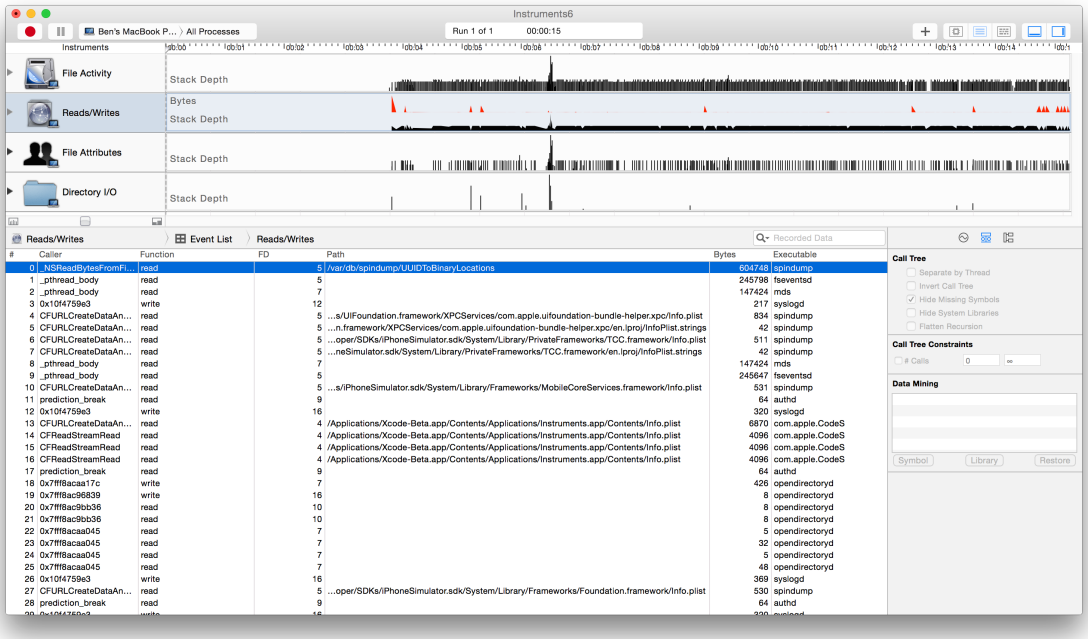
> **Tip:** After selecting a template in the list, hold down the Option key to change the Choose button to be the Profile button. Click the Profile button to create the document and automatically begin profiling the target process.

Collecting and Analyzing Data: The Trace Document Window

A trace document is a self-contained space for collecting and analyzing trace data. You use the document to organize and configure the instruments needed to collect data, and you use the document to view the data you've gathered, at both a high and low level.

Figure 2-2 shows a typical trace document after data collection is complete. Because a trace document window presents a lot of information, it has to be well organized. As you work with trace documents, information generally flows from left to right. The farther right you are in the document window, the more detailed the information becomes.

Figure 2-2 The Instruments trace window after recording data



About the Instruments Trace Document in *Instruments Help Articles* lists the key elements that are called out in the figure above, as well as their components, and provides a more in-depth discussion of how you use these features.

Adding and Configuring Instruments

The Instruments analysis tool uses trace documents to define sets of individual instruments to collect data and display that data to the user. Although there is no theoretical limit to the number of instruments you can include in a trace document, most documents contain fewer than ten for performance reasons.

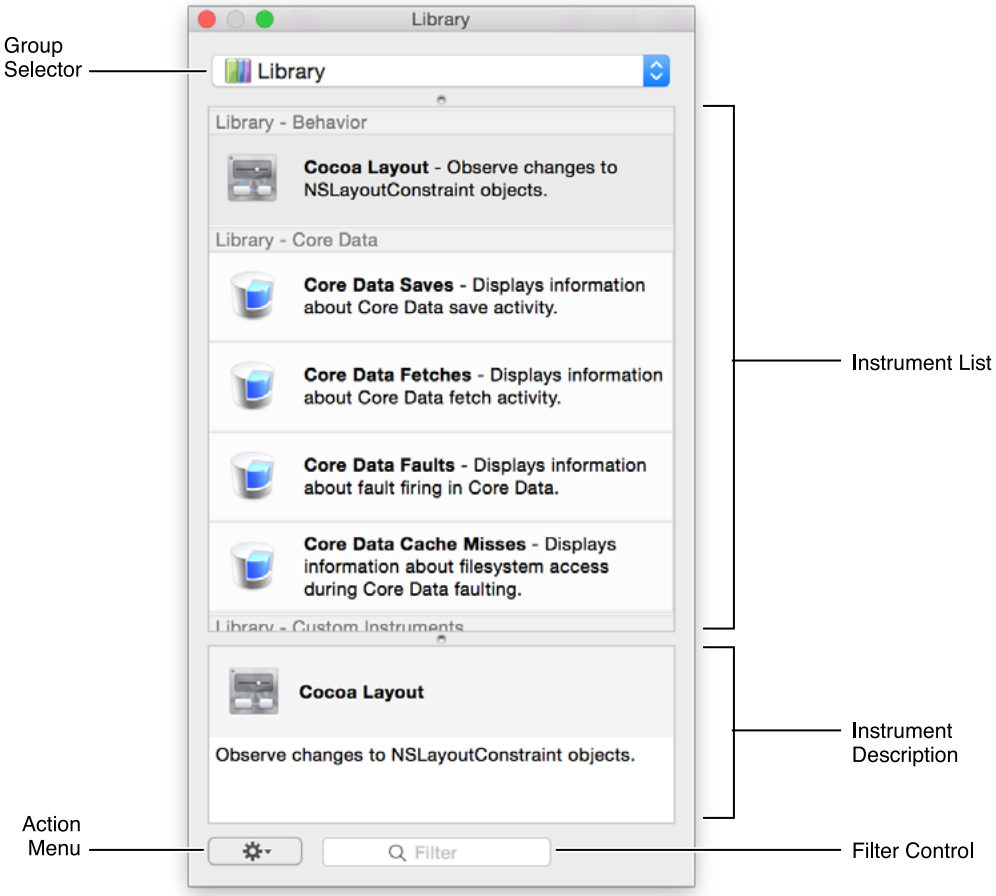
Instruments comes with a wide range of built-in instruments designed to gather specific data from one or more processes. Most of these instruments require little or no configuration to use. You simply add them to your trace document and begin gathering trace data. You can also create custom instruments, however, which provide you with a wide range of options for gathering data.

This chapter focuses on how to add existing instruments to your trace document and configure them for use. For information on how to create custom instruments, see Creating Custom Instruments (page 121).

Viewing the Built-in Instruments with the Library Window

The Library palette (shown in Figure 3-1) displays all of the instruments you can add to your trace document. The library includes all of the built-in instruments, in addition to any custom instruments you may have defined.

Figure 3-1 The instrument library



To open the Library window

Do one of the following:

- Click the Library (+) button in the trace document toolbar.
- Choose Window > Library.
- Press Command+L on your keyboard.

Because the list of instruments in the Library window can get fairly long—especially if you add your own custom-built instruments—the instrument library provides several options for organizing and finding instruments. The following sections discuss these options and show how you use them to organize the available instruments.

Changing the Library View Mode

The library provides different view modes to help you organize the available instruments. View modes let you choose the amount of information you want displayed for each instrument and the amount of space you want that instrument to occupy. Instruments supports the following view modes:

- **View Icons**: Displays only the icon representing each instrument.
- **View Icons And Labels**: Displays the icon and name of each instrument.
- **View Icons and Descriptions**: Displays the icon, name, and full description of each instrument.
- **View Small Icons And Labels**: Displays the name of the instrument and a small version of its icon.

> **Note:** Regardless of which view mode you choose, the Library window always displays detailed information about the selected instrument in the description area.

To change the library's view mode, select the desired mode from the action menu at the bottom of the Library window.

In addition to changing the library's view mode, you can display the parent group of a set of instruments by choosing Show Group Banners from the Action menu. The Library window organizes instruments into groups, both for convenience and to help you narrow down the list of instruments you want. By default, this group information is not displayed in the main pane of the Library window. Enabling the Show Group Banners option adds it, making it easier to identify the general behavior of instruments in some view modes.

Finding an Instrument in the Library

By default, the Library window shows all available instruments. Each instrument, however, belongs to a larger group, which identifies the purpose of the instrument and the type of data it collects. To limit the number of instruments displayed by the Library window, use the group selection controls at the top of the Library window to select fewer groups. When there are many instruments in the library, this feature makes it easier to find the instruments you want.

The group selection controls have two different configurations. In one configuration, the Library window displays a pop-up menu, from which you can select a single group. If you drag the split bar between the pop-up menu and the instrument pane downward, however, the pop-up menu changes to an outline view. In this configuration, you can select multiple groups by holding down the Command or Shift key and selecting the desired groups.

Figure 3-2 shows the Library window's standard mode, in which you select a single group using the pop-up menu. Figure 3-3 shows outline mode, in which you can select multiple groups and manage your own custom groups.

Figure 3-2 Viewing an instrument group in standard mode

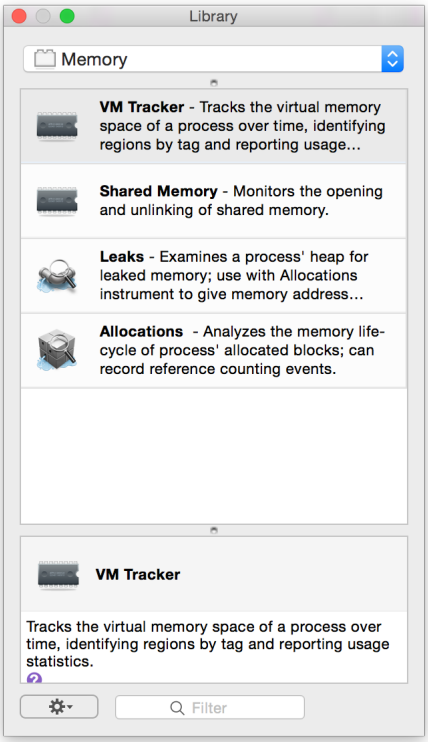
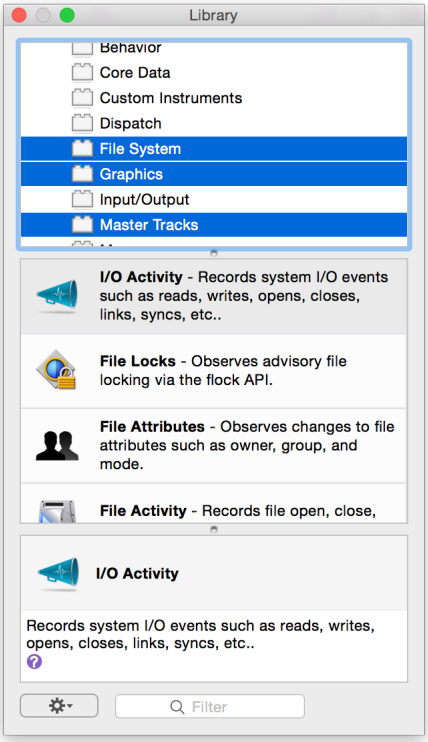
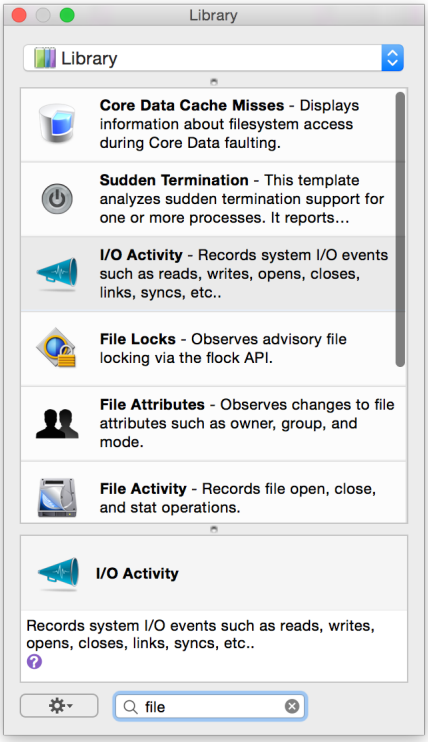


Figure 3-3 Viewing an instrument group in outline mode



Another way to filter the contents of the Library window is to use the search field at the bottom of the Library palette. Using the search field, you can quickly narrow down the contents of the Library to find any instruments whose name, description, category, or list of keywords matches the search text. For example, Figure 3-4 shows instruments that match the search string `file`.

Figure 3-4 Searching for an instrument



When the pop-up menu is displayed at the top of the Library window, the search field filters the contents based on the currently selected instrument group. If the outline view is displayed, the search field filters the contents based on the entire library of instruments, regardless of which groups are selected.

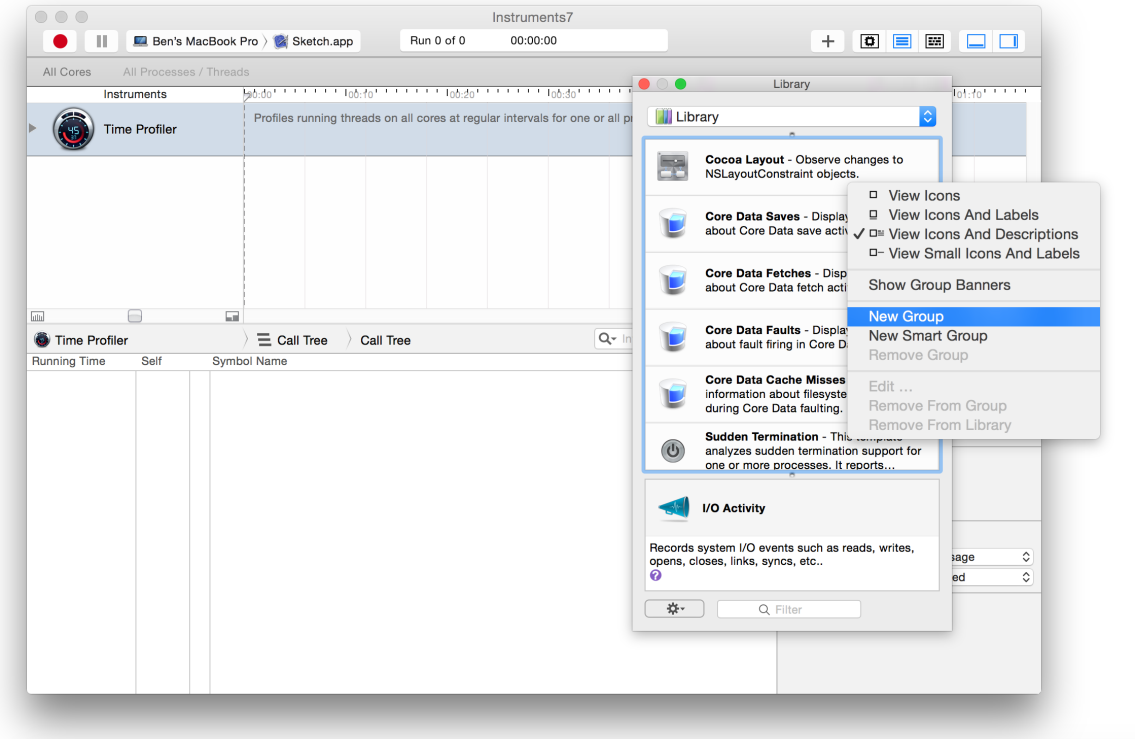
Creating a Custom Group

Create a group of instruments to customize and streamline your data gathering efforts.

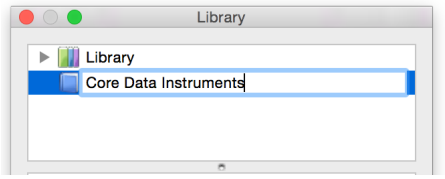
To create a static group in the Library window

1. Click the Library button (+) to open the Library window.

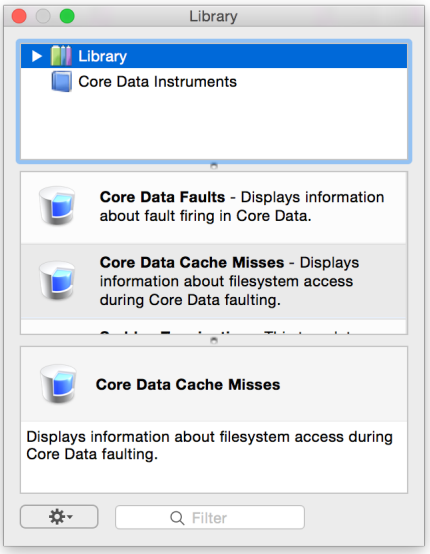
2. Control-click in the Library window and choose New Group or choose New Group from the action menu at the bottom of the Library window.



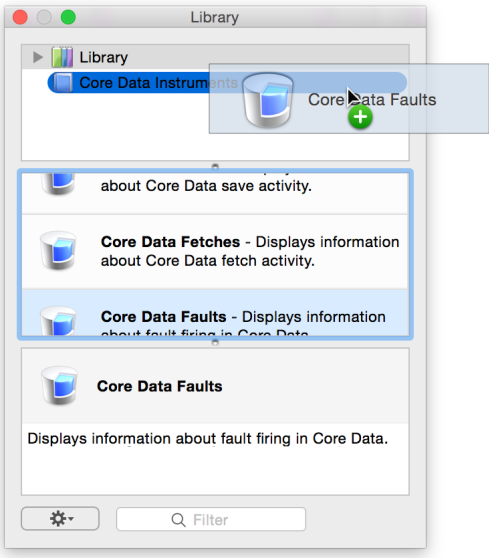
3. Enter a name for the group, and press Return.



4. Select the Library directory.



5. Find the desired instruments and drag them into the new group.



When you first open the Library window, the Library group and all of your custom groups are displayed. The Library group contains a complete list of all instruments. Populate a custom group with any number of instruments from the Library group. Give a custom group a name that identifies its purpose so you can easily select it when needed.

If an instrument has already been put into a group, the group name does not become highlighted if you try to drag the same instrument into it a second time.

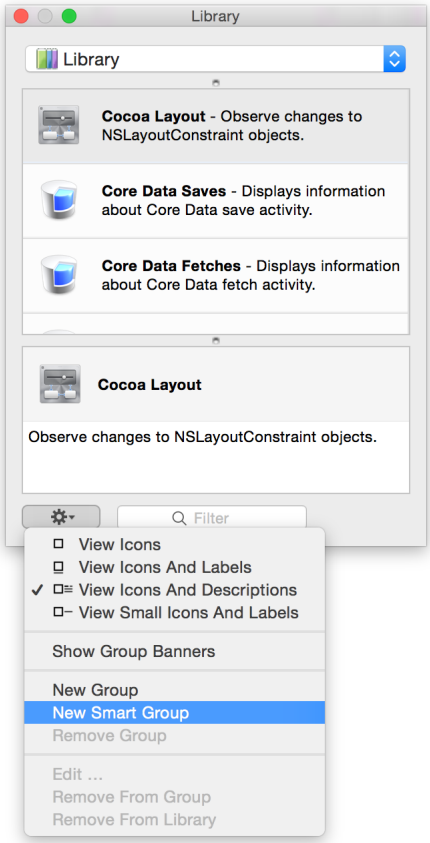
You can nest groups in a hierarchical structure by dragging them into one another. You can also drag a nested group outside the structure to stand alone.

Creating a Smart Group

Smart groups are instruments that have been grouped together based on a set of user-created rules.

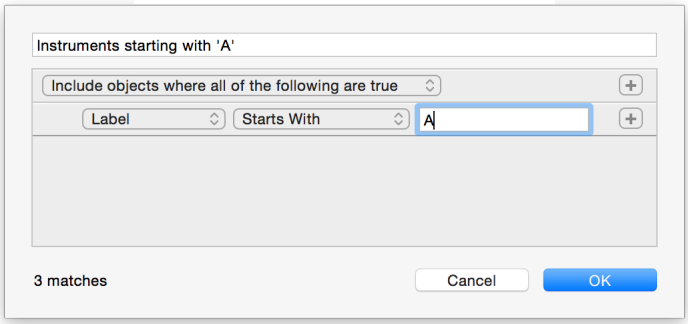
To create a smart group

1. Control-click in the Library window and choose New Smart Group or choose New Smart Group from the action menu at the bottom of the Library window

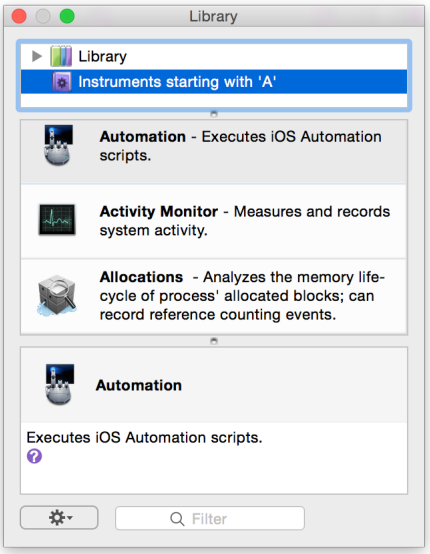


2. Enter a name for the group in the Label field.
3. Specify a rule for the group.

4. Click the Add button (+) to create a new rule.



5. Click OK.
6. Verify that the smart group populated correctly.



Every smart group must have at least one rule. You can add additional rules, as needed, using the controls in the rule editor and then configuring the group to apply all or some of the rules. Table 3-1 lists the criteria you can use to match instruments.

Table 3-1 Smart group criteria

Criteria	Description
Label	Matches instruments based on their title. Available comparison operators include Starts With, Ends With, and Contains.

Criteria	Description
Used Within	Matches instruments based on when they were last used. You can use this filter for instruments that were used within the last few minutes, hours, days, or weeks.
Search Criteria Matches	Matches instruments whose title, description, category, or keywords include a specified string.
Category	Matches instruments whose library group name contains or exactly matches the specified string. This criteria does not match against custom groups.

To edit an existing smart group, select the group in the Library window and choose Edit *groupname* from the action menu, where *groupname* is the name of your smart group. Or, Control+click on the group to access this menu. Instruments displays the rule editor again so you can modify the existing rules.

To remove a smart group from the Library window, select the group and choose Remove Group from the action menu or Control+click on the group and choose Remove Group. If you are currently viewing groups using the outline view, you can also select the group and press the Delete key.

Adding an Instrument to a Trace Document

Expand on the amount and type of data collected in a trace document by adding new instruments.

To add an instrument from the Library window

1. Open the Library window.
2. Select Library from the pop-up at the top of the palette to display a list of all instruments. Or, select a specific category to display related instruments. You can also use the Filter field at the bottom of the Library palette to narrow down the list.
3. Drag an instrument to the desired position in the instruments pane, or double-click on the instrument to add it to the bottom of the list.

Configuring an Instrument

Most instruments are ready to use as soon as you add them to the Instruments pane in a trace document. Some instruments can also be configured in the Record Settings area of the inspector sidebar. Customizable settings vary from instrument to instrument. Most instruments contain options for configuring the contents of the track pane, and a few contain additional controls for determining what type of information is gathered by the instrument itself.

To open the configuration options for an instrument, select the instrument in the track view and then press Command-1 (or click the Record Settings button in the inspector sidebar).

Controls that adjust the display of information in the track pane can be configured before, during, or after your record data for the track. While Instruments automatically gathers the data it needs for many display options, regardless of whether that option is currently displayed in the track pane, some options may not take effect until the next run.

For more information about the inspector sidebar, see About the Instruments Trace Document.

Collecting Data on Your App

For Instruments to help you monitor and improve your app, it has to be able to collect information on your app while it is running. This chapter describes how to direct Instruments to collect information on your app.

> **Note:** In order to use Instruments to profile an iOS device, your device must be provisioned for development before data can be collected from it. See Provisioning Your iOS Device for Development.

Setting Up Data Collection with the Target Pop-up Menu

The target pop-up menu in the toolbar is used to set both the device and the app or process on which you will be collecting data. Click on the target pop-up menu to make your choice.

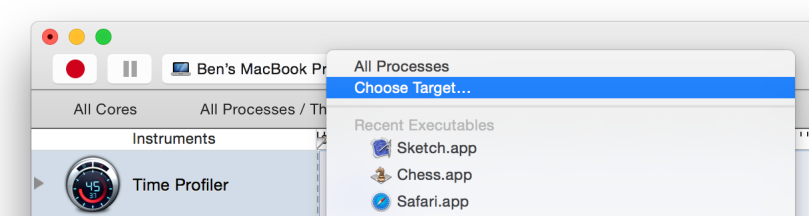
The Target pop-up menu provides you with the following main choices for collecting data:

- **All Processes.** Collects data from all processes currently running on the system.
- **Choose Target.** Collects data from a specific app that you specify. The app automatically launches when you click the Record button.

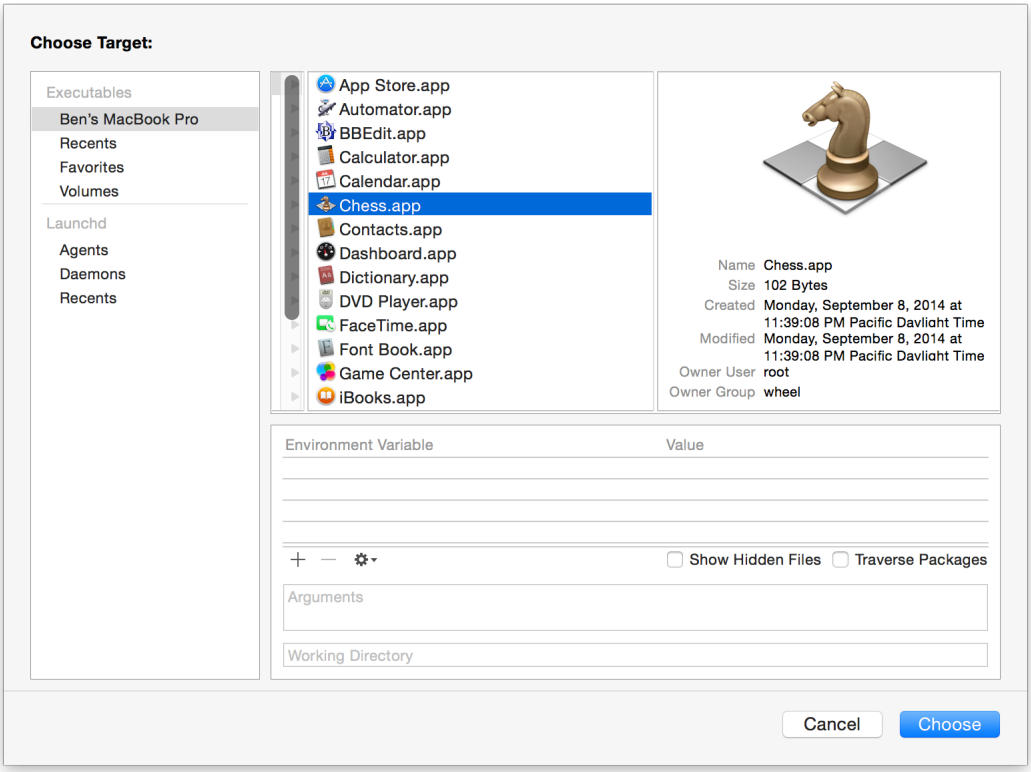
Typically, users collect data from one app at a time. When you target a single app, all of the instruments in the trace document collect data from the targeted app.

To target a single app in a trace document

1. Select Choose Target from the target pop-up menu.



2. Locate your app and press the Choose button.



The target pop-up menu also provides quick access to select certain processes to target without opening the choose target window. For iOS development, this including apps you've installed, running apps, and system processes. For OS X development, it includes recently profiled apps, running apps, and system processes.

When selecting a platform in the target pop-up menu, you only see iOS devices when they are plugged into your computer or when you have configured Instruments to collect data from a particular device wirelessly.

To enable an iOS device for wireless profiling

1. Make sure your mobile device is connected to your development system by a USB cable.
2. Press the Option key and click the target pop-up menu in the toolbar of your trace document.
3. Choose your mobile device to enable the wireless option.
4. Open the target pop-up menu and choose the wireless version of your device.
5. Disconnect the device from the USB cable.

Using a wireless connection allows you to move your device as needed for testing without getting tangled in the cable or accidentally unplugging the device during testing. Connecting wirelessly is especially useful when testing the following:

- **Accelerometers.** Move the device in all directions without its being tethered. Connecting wirelessly ensures a complete testing of the device.
- **Accessories.** Plug your USB accessory into the free slot and test it.

> **Important:** Your device must be provisioned for development before Instruments can collect data from it.
>
> Bonjour and multicast must be enabled on your wireless network access point.
>
> Both the device and the recording Mac will need to be on the same wireless network and subnet.

> **Note:** Turning off the device causes data collection to stop. You must reconnect the device to your computer to resume data collection.

Collecting Data from the Dock

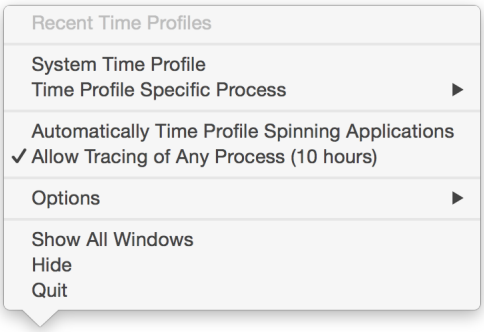
Save time by running Time Profiler from the Instruments app icon in the Dock to automatically record certain events in the background, even without Instruments running. Using this method, you can perform the following tasks:

- **System Time Profile.** Starts profiling all system processes.
- **Time Profile Specific Process.** Starts the Time Profiler instrument with a specific app from the pop-up menu.
- **Automatically Time Profile Spinning Applications.** Automatically profiles blocked (spinning) apps in the future.
- **Allow Tracing of Any Process (10 hours).** Trace any process that occurs in the next 10 hours. Bypasses having to type in a password during the 10 hours.

To collect Time Profiler information from the Dock

1. With Instruments opened, Control-click the Instruments dock icon.

2. Choose the process to profile to start recording.



Collecting Data Using iprofiler

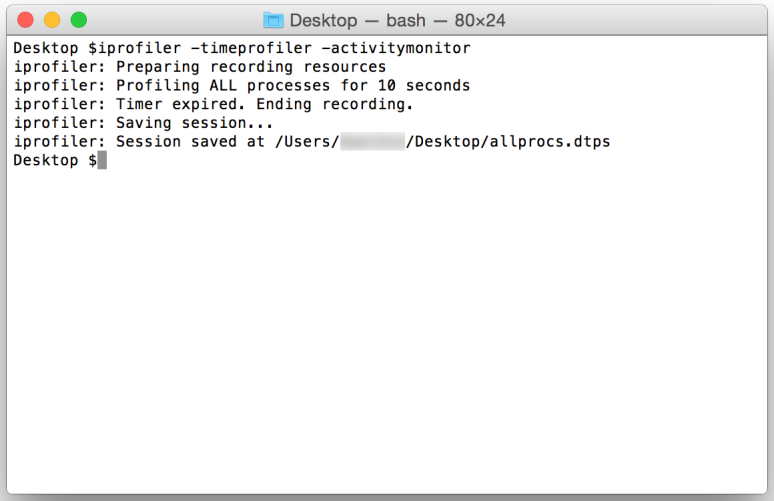
`iprofiler` is a command-line tool used to measure an app's performance without launching Instruments. After collecting the performance data, import the data to Instruments in order to get a visual representation of the data. The collected data is saved in a `.dtps` bundle that can be opened by Instruments. `iprofiler` supports the following trace templates:

- **Activity Monitor.** Monitors overall system activity and statistics, including cpu, memory, disk, and network. Activity Monitor also monitors all existing processes and parent/child process hierarchies.
- **Allocations.** Measures heap memory usage by tracking allocations, including specific object allocations by class. Allocations can also record virtual memory statistics by region.
- **Counters.** Collect performance monitor counter events using time or event based sampling methods.
- **Event Profiler.** Samples the processes running on the system's CPUs through low-overhead, event-based sampling.
- **Leaks.** Measures general memory usage, checks for leaked memory, and provides statistics on object allocations by class as well as memory address histories for all active allocations and leaked blocks.
- **System Trace.** Provides comprehensive information about system behavior by showing when threads are scheduled, and showing all their transitions from user into system code through either system calls or memory operations.
- **Time Profiler.** Performs low-overhead, time-based sampling of processes running on the system's CPUs.

To collect and view data from iprofiler

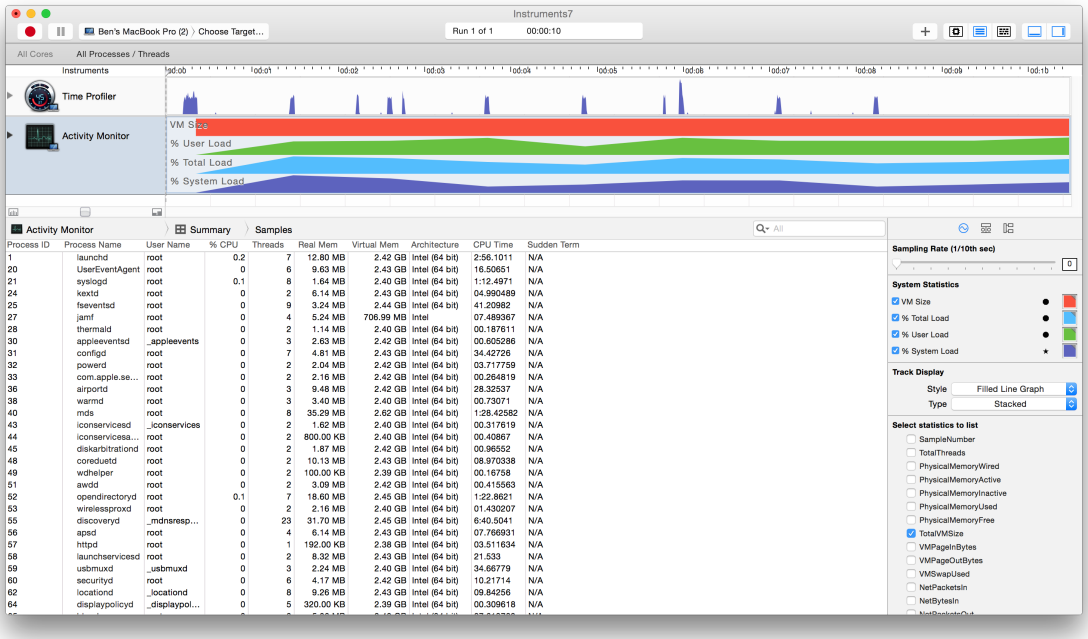
1. Open Terminal.

2. Enter an `iprofiler` command to collect data.



3. Open Instruments and click File > Open.
4. Find the saved `.dtps` file and click Open.

After importing the saved file, Instruments automatically opens the associated instruments in a trace document and populates them with the collected data. You can view and manipulate this data to locate any issues with your app.



`iprofiler` provides a limited set of configuration options for defining what data to collect.

Command	Description
`-l`	Provides a list of all supported instruments.
`-L`	Provides a list of all supported instruments and a description of what each template does.
`-legacy`	Executes the legacy Instruments command-line interface found in `/usr/bin/instruments`.
`-T duration`	Sets the length of time that data is collected for. If this option is not set, then data is collected for 10 seconds. Duration can be set to seconds (1s or 1), milliseconds (10m or 10ms), or microseconds (10u or 10us).
`-I interval`	Sets the frequency with which a measurement is taken during the sample time. If this option is not specified, it uses the Instruments default interval time. The interval can be set to seconds (1s or 1), milliseconds (10m or 10ms), or microseconds (10u or 10us).
`-window period`	Limits the performance measurement to the final period of the `iprofiler` run. If this option is not specified, performance is measured throughout the entire run. The period can be set to seconds (1s or 1), milliseconds (10m or 10ms), or microseconds (10u or 10us). **Note:** This option can be used only with the `-timeprofiler` and `-systemtrace` template options.
`-d path -o basename`	Specifies the destination path and the name used when saving the collected data. If the path is not specified, the output is saved to the current working directory. If the basename is not specified, the process name or process ID is used.
`-instrument name`	Designates the instrument that is to be run. Valid name options are `-activitymonitor`, `-allocations`, `-counters`, `-eventprofiler`, `-leaks`, `-systemtrace`, and `-timeprofiler`. At least one template must be listed. You can run up to all seven templates at once.
`-kernelstacks`	Backtraces only include kernel stacks when this option is used. If neither `-kernelstacks` or `-userandkernelstacks` options are specified, backtraces include user stacks only.

Command	Description
`–userandkernelstacks`	Backtraces include both kernel and user stacks. If neither `–kernelstacks` or `–userandkernelstacks` options are specified, backtraces include user stacks only.
`–pmc PMC_MNEMONIC`	Use this flag with -counters to specify the mnemonic of the event to count. Multiple mnemonics should be comma-separated.
`–allthreadstates`	Causes the Time Profiler template to profile all threads. If this is not specified, then Time Profiler profiles only running threads.
`–a process/pid`	Attaches to a process that is already running. Specifying a string will attach the process whose name starts with that string. Specifying a process ID attaches it to the process associated with that process ID. The `–leaks` option requires that a specific single process or process ID be specified.
`executable [args…]`	Causes the target process to be launched for the duration of the measurement. Lists the executable and the arguments as if they are being invoked from the command-line.

iprofiler Examples

A list of common `iprofiler` command-line examples are below.

The following example collects data from all running processes for the current sampling duration set in Instruments using the Time Profiler and Activity Monitor instruments. The collected data is saved to the working directory as `allprocs.dtps`.

```
iprofiler –timeprofiler –activitymonitor
```

The following example opens and collects data from YourApp using the Time Profiler instrument. Data is collected for eight seconds and the data is saved at `/temp/YourApp_perf.dtps`.

```
iprofiler –T 8s –d /temp –o YourApp_perf –timeprofiler –a YourApp
```

The following example collects data from the process with the 823 process ID using the leaks and activity monitor instruments. Data is collected for 2500 milliseconds (2.5 seconds) and saves it to the working directory as `YourApp_perf.dtps`.

```
iprofiler –T 2500ms –o YourApp_perf –leaks –activitymonitor –a 823
```

The following example opens and collects data from YourApp using the Time Profiler and Allocations instruments. Data is collected for the default amount of time set in Instruments and saved in `/tmp/allprocs.dtps`.

```
iprofiler -d /tmp -timeprofiler -allocations -a YourApp.app
```

The following example opens and collects data from YourApp found in `/path/to` with the argument `arg1` using the Time Profiler and System Trace instruments. Data is collected for fifteen seconds, but only the data collected in the last 2 seconds is saved. The data is saved to the working directory as `YourApp_perf.dtps`.

```
iprofiler -T 15 -I 1000ms -window 2s -o YourApp_perf -timeprofiler -systemtrace
/path/to/Your.app arg1
```

Minimizing Instruments Impact on Data Collection

Instruments is designed to minimize its own impact on data collection. By changing some basic settings, you can further decrease the impact Instruments has on data collection.

You can decrease the sample interval for many instruments in order to collect more data. However, high sample rates that result from a short sample interval can cause several problems:

- **Processor time is required for every sample.** High sample rates use more processor time.
- **Sample interval timing may not be consistent.** Interrupts are used to start each sample. Variables in when these interrupts occur can cause significant variations in the sample rate when using very small sample intervals.
- **Small sample intervals cause more samples to be taken.** Each sample uses system memory and a large number of samples quickly uses up the available memory on smaller memory machines.

You can change the sample interval for an instrument by opening the Record Settings inspector for that instrument.

Running Instruments in Deferred Mode

Increase the accuracy of performance-related data by deferring data analysis until you quit the app you are testing. Typically, Instruments analyzes and displays data while your app runs, allowing you to view the data as it is collected. Performing analysis live slows down the target process by taking up CPU time and memory, which leaves you with measurements that may not reflect how the process would normally behave. Running

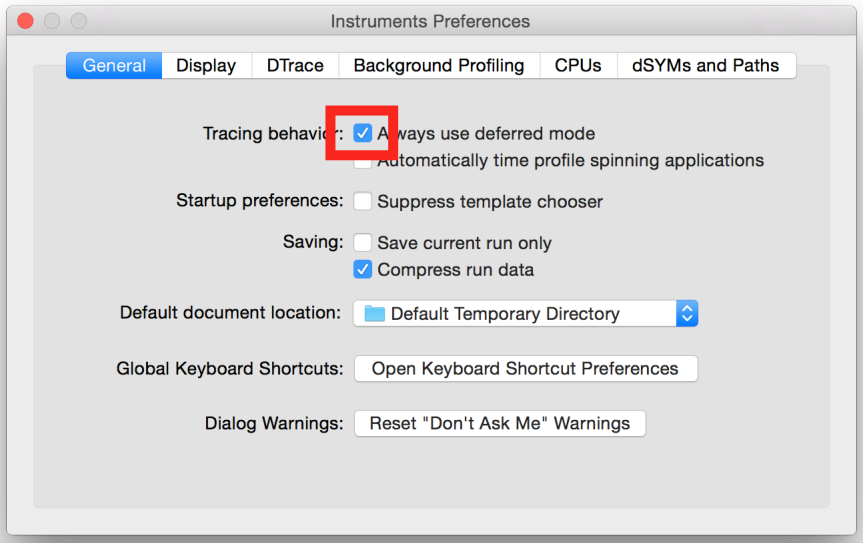
Instruments in deferred mode delays the analysis of data until the data collection is done, either after your app has finished running or after you click Stop. While in deferred mode, you are blocked from interacting with the instruments that are collecting data.

In deferred mode, after Instruments has finished collecting data, Instruments processes the data and displays it onscreen. Even though deferring the data analysis adds time to the later stages of the data collection process, it helps ensure that performance-related data is accurate.

You can set Instruments to run in deferred mode in the Instruments preferences.

To set deferred mode for Instruments

1. Choose Instruments > Preferences.
2. In the General tab, select the "Always use deferred mode" checkbox.

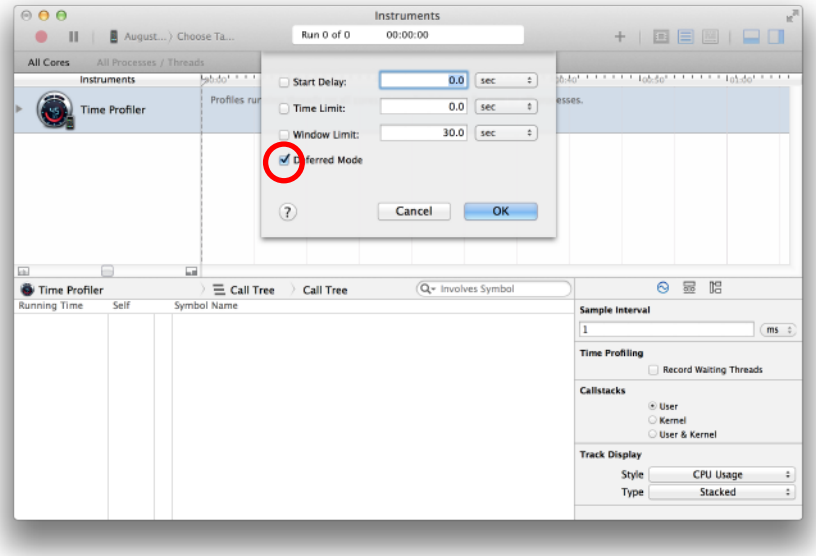


Using deferred mode moves the data processing to after the collection of data, resulting in a delay at the end of data collection as Instruments processes the data. In the event of especially long traces, this delay can be significant. To avoid this delay, you can set deferred mode for only those traces that require extremely precise data collection.

To set deferred mode for a trace

1. Choose File > Record Options.

2. Select the Deferred Mode checkbox, then click OK.



Examining Your Collected Data

After collecting information about your app, you need to examine what you collected. Even though every instrument is different, they have several things in common. This chapter describes common tasks that are used to help you examine the information you have collected.

Locating Symbols for Your Data

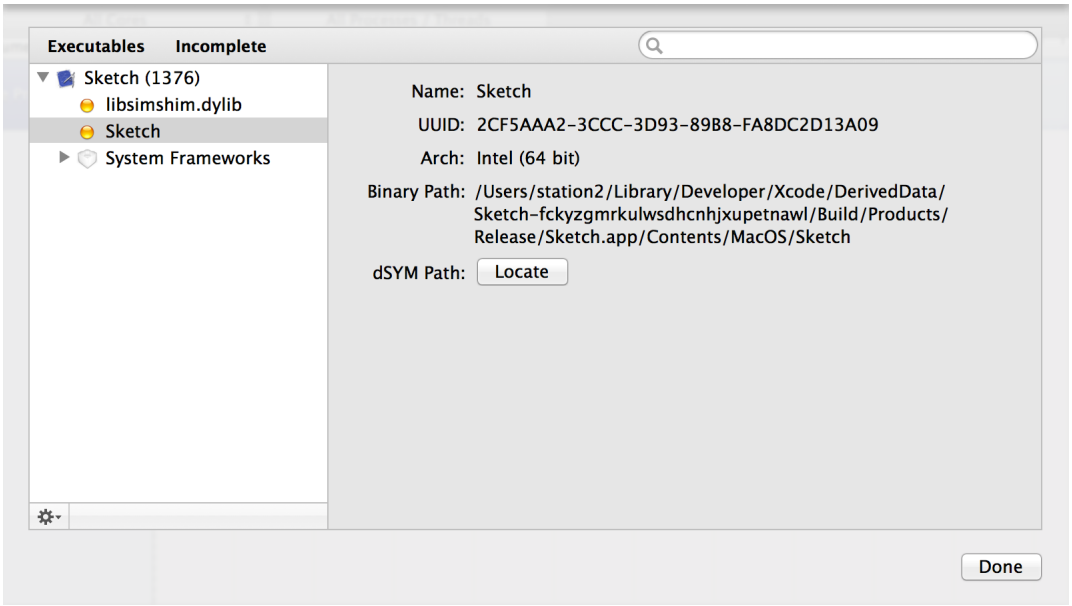
Instruments requires accurate information about your project in order to provide you with the best results. You will get the most complete information if the system can see all of the symbols associated with your project. When addresses, rather than symbols, are displayed in trace documents generated by the Instruments analysis tool, you can manually provide the missing information. The mapping of addresses to their symbols is contained in a dSYM file. Typically, Instruments finds dSYM files automatically, based on the locations that Spotlight indexes and the paths you specify in Instruments Preferences. However, you can point Instruments in the right direction when it can't automatically find a dSYM. Once you do, Instruments can map addresses to their associated symbols and line number information.

To locate the dSYM path for a trace document

1. Choose File > Symbols.

   This option appears only after running a trace or loading a previously saved trace.

2. Select the executable (binary) or framework that is missing symbols.



3. Click the Locate button; the “Select dSYM or containing folder” dialog opens.
4. Select your symbol file or the containing folder.
5. Click Open.

To correctly display symbols in trace documents, Instruments needs access to the specific symbol files that were generated when the executable you are testing was built. Therefore, you may need to manually locate a dSYM file when you build on one machine and test performance on another.

The filter bar in the Symbols dialog lets you filter by binary/framework or missing information, or you can search for a binary/framework by entering text into the search field.

Note: By default, Xcode saves symbols as dSYM bundles in the Build folder. Xcode creates the default dSYM bundle path by adding the extension `.dSYM` to the executable name. To generate dSYM bundles in Xcode, go to the Build Settings area for your project and in the Build Options section, select Debug Information Format > Dwarf with dSYM File.

Caution: Normally, when an address is displayed instead of a symbol, the address appears in black on the left in the detail pane and the owning library in gray on the right. You can identify these addresses by locating the appropriate dSYM file.

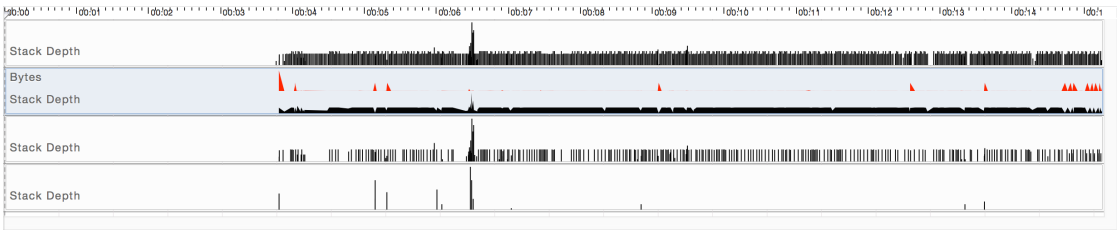
If the address is gray and no owning library is listed, you cannot display the symbolic name. A gray address with no library name means that at the time the trace was recorded, you did not have permission to see information about that process. This typically occurs when the Code Signing Identity in your build settings is set to a release/distribution identity for Release builds. Your Release build should use the same setting as your Debug build (typically, "iOS Developer").

Viewing the Collected Data in the Track Pane

The most prominent portion of a trace document window is the track pane. The track pane occupies the area immediately to the right of the instruments pane. This pane presents a high-level graphical view of the data gathered by each instrument. You use this pane to examine the data from each instrument and to select the areas you want to investigate further.

The graphical nature of the track pane makes it easy to spot trends and potential problem areas in your app. For example, a spike in a memory usage graph indicates a place where your app is allocating more memory than usual. This spike might be normal, or it might indicate that your code is creating more objects or memory buffers than you had anticipated in this location. An instrument such as the Leaks instrument can also point out places where your app is not handling memory properly. If the graph for Leaks is relatively empty, you know that your app is behaving properly, but if the graph is not empty, you might want to examine why that is. Figure 5-1 shows an example of the track pane after collecting data.

Figure 5-1 An example of graphical data in the track pane of a trace document



Use the timeline at the top of the pane to select the period you want to investigate. Clicking in the timeline moves the *playhead* (a position control showing a common point in time) to that location and displays a set of inspection flags that summarize the information for each instrument at that location. Clicking in the timeline also focuses the information in the Detail pane on the surrounding data points; see Examining Data in the Detail Pane (page 50).

Although each instrument is different, nearly all of them offer options for changing the way data in the track pane is displayed. In addition, many instruments can be configured to display multiple sets of data in their track pane. Both methods give you the ability to display collected data in a way that makes sense for your app.

The sections that follow provide more information about the track pane and how you configure it.

Setting Flags

Flags allow you to quickly access points of interest in the track pane. You can add names and descriptions to flags in order to add information specific to that flag.

To set flags at the current playhead position in the track pane

Do one of the following:

- Choose Edit > Add Flag.
- Press Command–Down Arrow.

Zooming In and Out

After data has been recorded, you can zoom in and out on the track pane to expand or contract the detail presented. Dragging your mouse cursor across a section of data allows you to zoom in or out when done with either the Shift or Control keys pressed. The zoom slider also controls the level of zoom in the track pane.

To zoom in and out of your data

Do one of the following:

- To zoom in, press the Shift key, and drag across the section of data you wish to isolate. Alternately, you can drag the zoom slider to the right.
- To zoom out, press the Control key, and drag across a section of data. Alternately, drag the zoom slider to the left.
- Drag the track pane zoom slider at the bottom of the Instruments pane.

For more information, see Zooming In, Zooming Out, and Isolating a Segment of the Data Collection Graph.

Viewing Data for a Specific Range of Time

Zooming in on a particular event in the track pane lets you see what happened around a specific time. You may also want to see only the data collected over a specific range of time.

You can set an inspection range by holding the mouse button down and dragging in the track pane of the desired instrument. If you drag across a section of data without using any modifier keys, you will select that data to be the only data displayed. Dragging makes the instrument under the mouse the active instrument (if it is not already) and sets the range using the mouse-down and mouse-up points.

You can also use the Inspection Range controls in the View menu to accurately select only the data collected in a specific time range.

To select a time range for inspection using the View menu

1. Set the start of the range.
    a. Drag the playhead to the desired starting point in the track pane.
    b. Click View > Set Inspection Range Start.
2. Set the end of the range.
    a. Drag the playhead to the desired endpoint in the track pane.
    b. Click View > Set Inspection Range End.

Instruments highlights the contents of the track pane that fall within the range you specified. When you set the starting point for a range, Instruments automatically selects everything from the starting point to the end of the current trace run. If you set the endpoint first, Instruments selects everything from the beginning of the trace run to the specified endpoint.

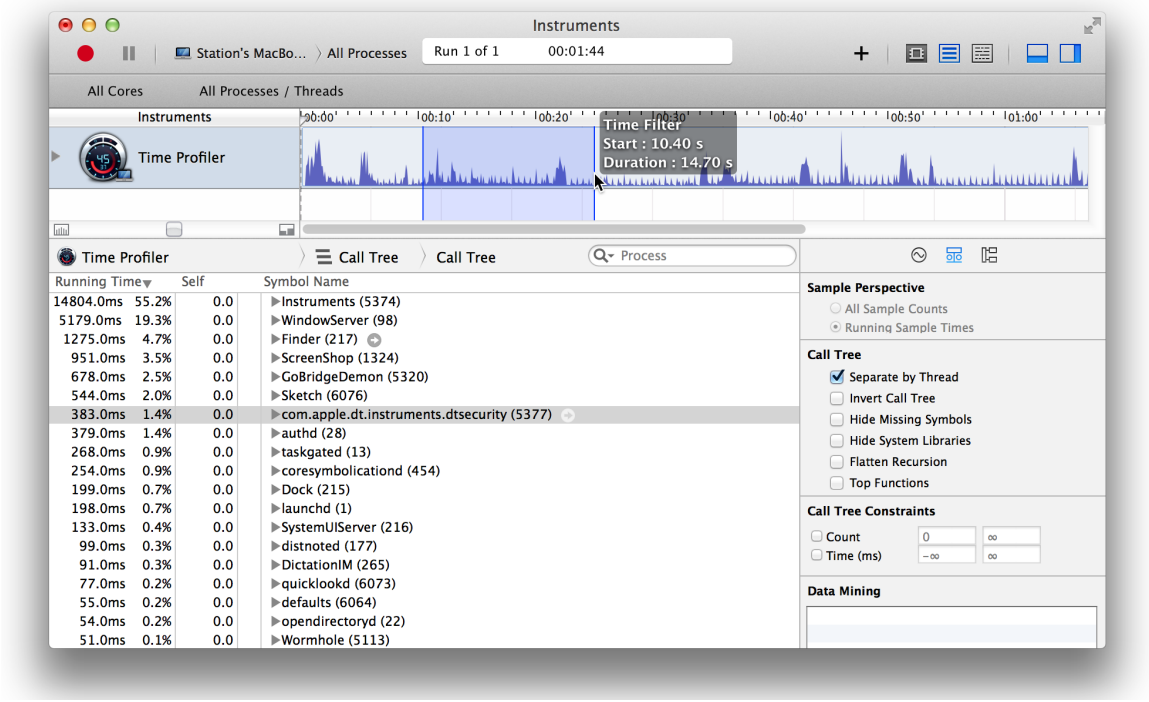
When you set a time range, Instruments filters the contents of the detail pane, showing data collected only within the specified range. You can use this method to very quickly narrow down a large amount of information collected by Instruments and see only the events that occurred over a certain period of time.

To clear an inspection range, click outside the selected range or click View > Clear Inspection Range.

Isolating a Segment of the Data Collection Graph

While pressing the Option key, drag across a section of the data collection graph to view the data in the selected range. As you drag the cursor, the start time and the duration of the time filter appear. The detail pane changes to display only the information contained within the time filter. Figure 5-2 shows a section of the track pane highlighted before it is zoomed in on.

Figure 5-2 Zooming in on a section of data

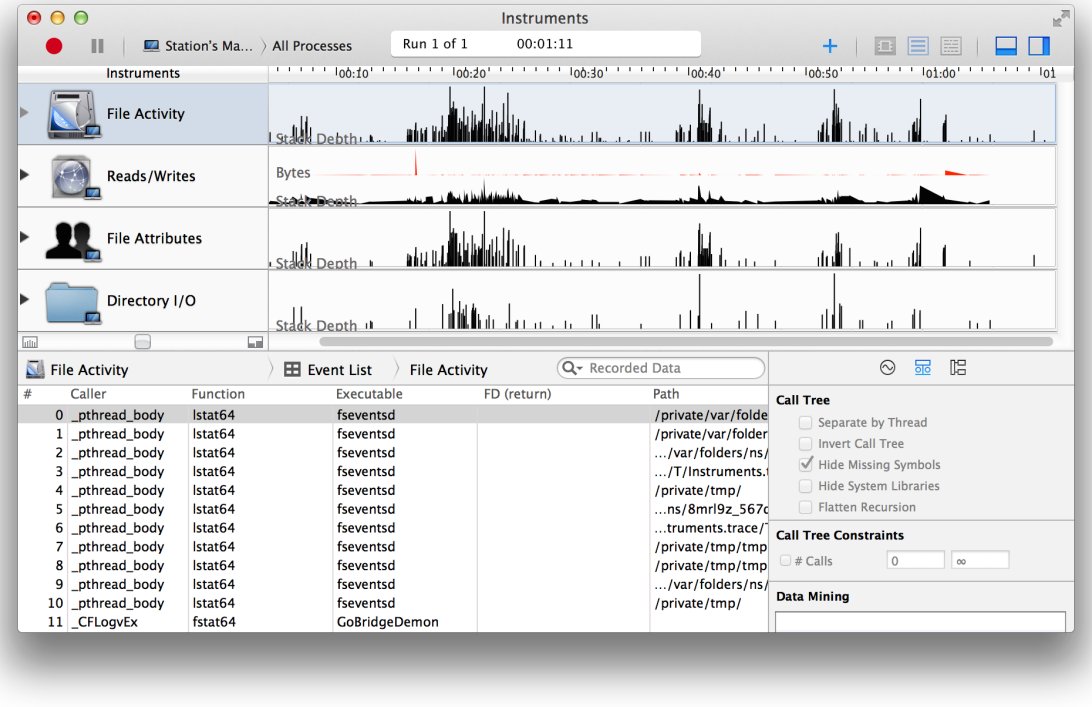


Examining Data in the Detail Pane

After you identify a potential problem area in the track pane, use the Detail pane to examine the data in that area. The detail pane displays the data associated with the current trace run for the selected instrument. Instruments displays only one instrument at a time in the detail pane, so you must select different instruments to see different sets of details.

Different instruments display different types of data in the detail pane. Figure 5-3 shows the detail pane associated with the File Activity instrument, which records information related to file system routines. The detail pane in this case displays the function or method that called the file system routine, the file descriptor that was used, and the path to the file that was accessed.

Figure 5-3 The detail pane



To open or close the Detail pane

Do one of the following:

- Choose View > Detail.
- Click the left View button at the right end of the toolbar.

Changing the Display Style of the Detail Pane

For some instruments, you can display the data in the detail pane using more than one format. For example, the Activity Monitor instrument allows you to view a summary of data, parent child information, and a list of samples. To view an instrument's data using one of the available formats, choose the appropriate mode from the rightmost menu in the navigation bar at the top of the detail pane. Which modes an instrument supports depends on the type of data gathered by that instrument.

For some modes, like the Time Profiler's call tree mode, you can use disclosure triangles in certain detail pane rows to dive further down into the corresponding hierarchy. Clicking a disclosure triangle expands or closes just the given row. To expand both the row and all of its children, hold down the Option key while clicking on a disclosure triangle.

Sorting in the Detail Pane

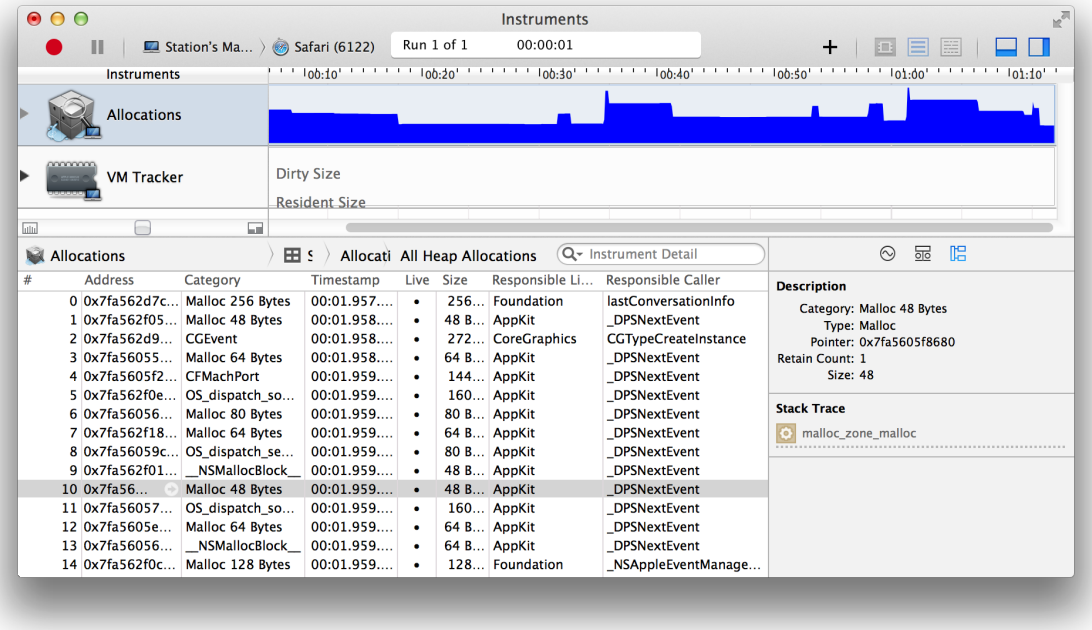
You can sort the information displayed in the detail pane according to the data in a particular column. To do so, click the appropriate column header. The columns in the detail pane differ with each instrument.

Working in the Extended Detail Inspector

For some instruments, the Extended Detail inspector shows additional information about the item currently selected in the detail pane. The Extended Detail inspector usually includes a description of the probe or event that was recorded, a stack trace, and the time when the information was recorded. Not all instruments display this information, however. Some instruments may not provide any extended details, and others may provide other information in this pane.

Figure 5-4 shows the Extended Detail inspector for the Allocations instrument. In this example, the instrument displays information about the type of memory that was allocated, including its type, pointer information, and size.

Figure 5-4 Extended Detail inspector



To open or close the Extended Detail inspector

Do one of the following:

- Choose View > Inspectors > Show Extended Detail.
- Press Command+2.
- Click the Extended Detail button in the Inspector control at the top of the Inspector sidebar pane.

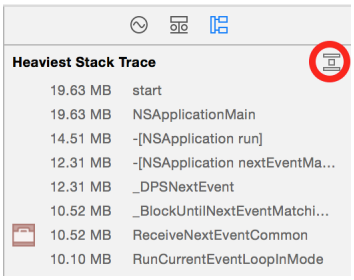
If you have an Xcode project with the source code for the symbols listed in a stack trace, you see the code in the detail pane's Console area. Instruments is able to display your code in the Detail pane and can even open Xcode so that you can make any desired changes.

To see your personal code

1. Click on a row in the detail pane.
2. Double-click the symbol name in the Extended Detail inspector.
3. Click the Xcode icon in the top right of the detail pane to make changes.

You can hide system calls in a stack trace by clicking the button at the top of the stack trace area in the Extended Detail inspector, as seen in Figure 5-5.

Figure 5-5 The button for hiding system calls in a stack trace



For more information on accessing source code from Instruments and viewing stack traces, please refer to Accessing Source Code from Instruments and Viewing Stack Traces Related to Your Code.

Saving and Importing Trace Data

Instruments provides several ways for you to save instrument and trace data for later use or reference. You can save data you've recorded in a trace document, or you can save the instrument configuration. Saving trace data lets you maintain a record of your app's performance over time. Saving configurations avoids the need to recreate commonly used configurations each time you run Instruments.

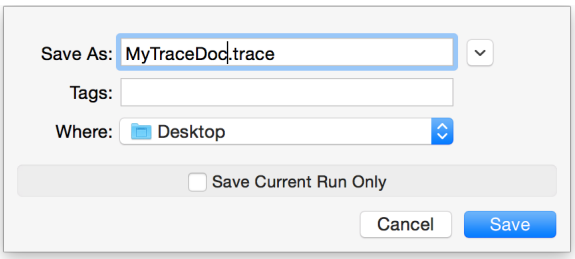
The following sections explain how to save your trace documents and how to export trace data to formats that other apps can read.

Saving a Trace Document

At times, you may want to save a set of instruments along with the data that they have collected over one or more trace sessions. Instruments saves the current document as an Instruments trace file, with the `.trace` extension.

To save a set of instruments

1. Select File > Save.
2. Enter a name for the file.
3. Enter a destination for the file.
4. To save trace data for the most recent run only, select the Save Current Run Only checkbox. If you've recorded multiple runs and want to save all of that data, then deselect this checkbox.
5. Click Save.



Tip: You can set the default state (selected or deselected) of the Save Current Run Only checkbox in the General pane of Instruments preferences.

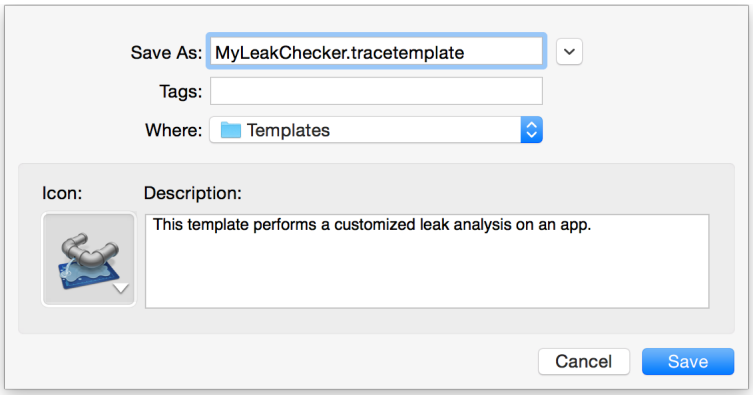
Saving an Instruments Trace Template

During your development cycle, you may need to gather trace data multiple times using a fixed set of instruments. Or, you may find yourself running the same set of instruments with the same configurations on multiple apps. Rather than setting up a new trace document and configuring it with the same set of instruments every time you run Instruments, you can build a trace document once and then save it as a trace template, allowing you to reuse it at anytime in the future.

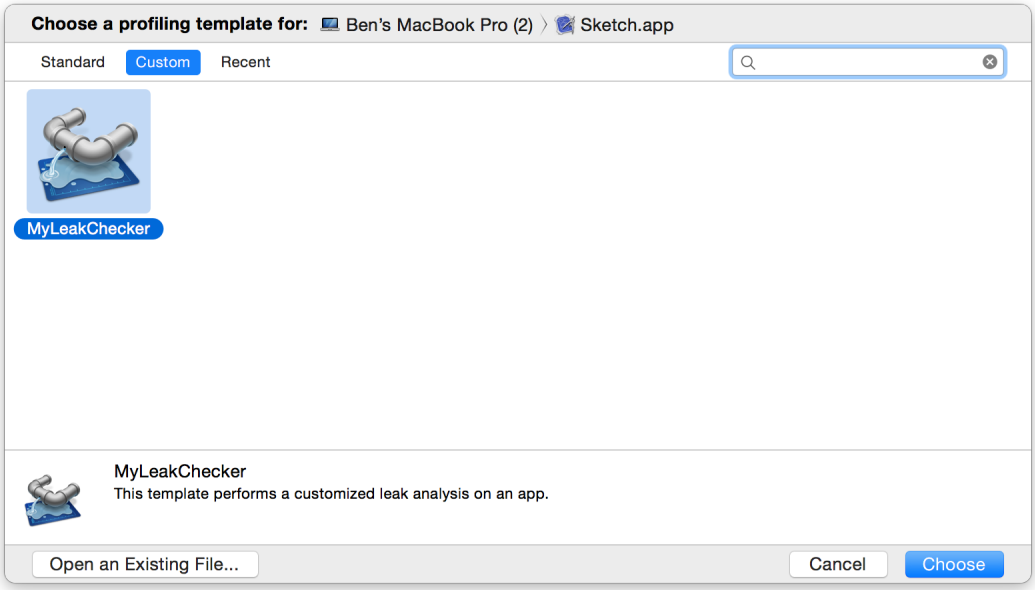
To save a trace template

1. Choose File > Save As Template.
2. Enter a name for the template.
3. Enter a destination for the template.

   Save your template in the `/Users/<username>/Library/Application Support/Instruments/Templates` directory to make it available in the custom section of the Instruments template window.
4. Select an icon for the template.
5. Enter a description for the template.
6. Click Save.



Once you've saved a template, you can open it in the same way you open other Instruments documents, by choosing File > Open. Or, if you saved it into the Instruments template folder, as mentioned above, you can open it from the template selection window.



When you open a trace template, Instruments creates a new trace document with the template configuration but without any data.

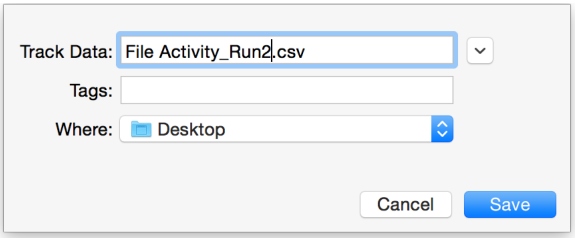
Exporting Track Data

Instruments allows you to export trace data to a comma-separated value (CSV) file format, a simple data file format that is is supported by many apps, including most database and spreadsheet apps. By importing data into another app, such as Numbers, you can manipulate it further, perform additional analysis, generate custom charts and graphs, compare it to other data, and more.

To export track data in CSV format

1. Select the instrument containing the data you want to export.
2. Choose Instrument > Export Track for '*<Instrument Name>*'
3. Enter a name for the file.
4. Enter a destination for the file.

5. Click Save.



Instruments exports the data for the most recent run of the selected instrument.

Note: Not all instruments support exporting to the CSV file format.

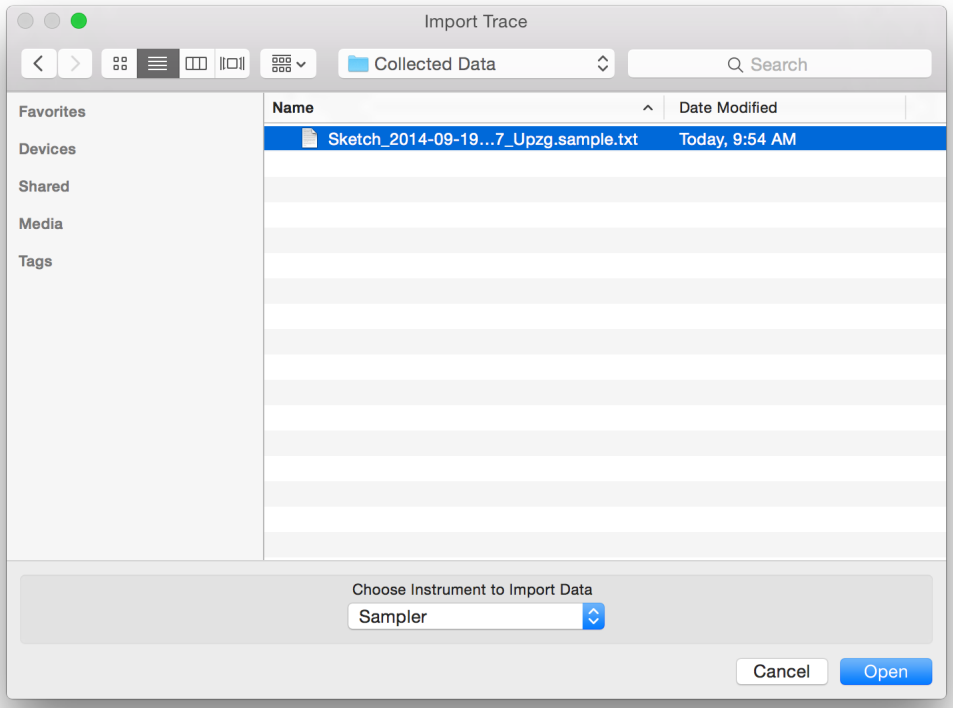
Importing Data from the Sample Tool

If you use the `sample` command-line tool to do a statistical analysis of your app's execution, you can import your sample data and view it using Instruments. Importing data from the `sample` tool creates a new trace document with the Sampler instrument and loads the sample data into the detail pane. Because the samples do not contain time stamp information, you can only view the data using outline mode in the detail pane. A new trace document is created based on the file you select.

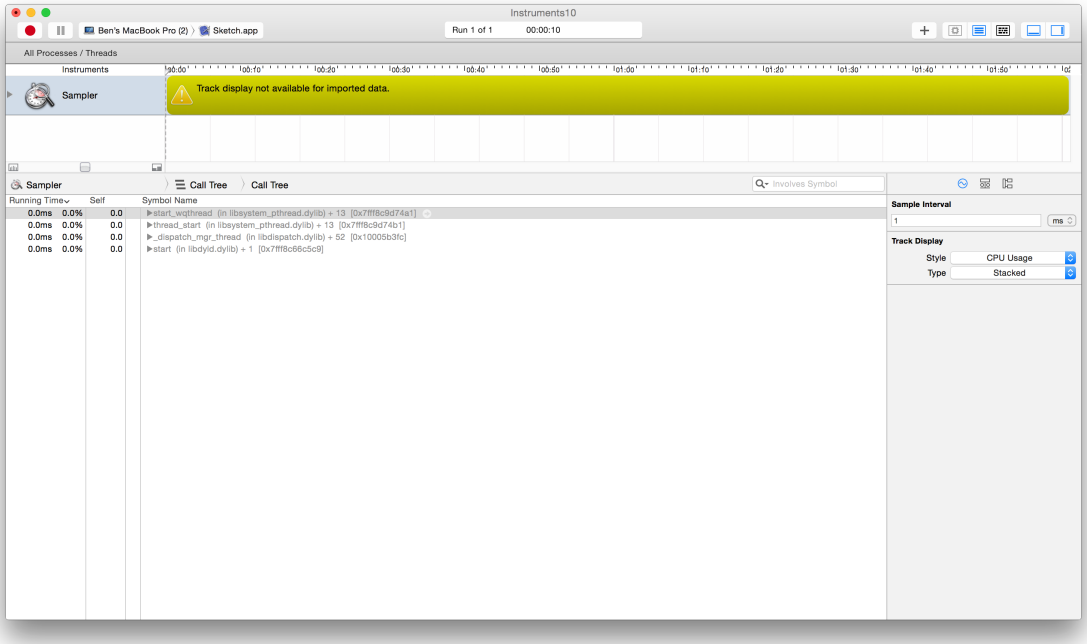
To import data from the Sample tool

1. Choose File > Import Data.
2. Locate your saved data.

3. Choose Sampler from the “Choose Instrument to Import Data” pop-up menu in the import window.



4. Click Open.



Working With DTrace Data

If your trace document contains custom instruments, you can export the underlying scripts for those instruments and run them using the `dtrace` command-line tool. After running the scripts, you can then reimport the resulting data back into Instruments. For information on how to do this, see Exporting DTrace Scripts (page 134).

Locating Memory Issues in Your App

Managing the memory that your app uses is one of the most important aspects of creating an app. From the smallest iOS device to the largest OS X computer, memory is a finite resource. For many developers, most of the memory size reduction they achieve in their apps and processes comes from using the Allocations instrument to see where most of the memory is being allocated and figuring out ways to reduce that size, perhaps by changing their data structures and/or algorithms. But even the best app designs can be plagued by memory issues that are difficult to isolate. This chapter describes how to identify several common memory issues, from memory leaks to zombies.

Examining Memory Usage with the Activity Monitor Trace Template

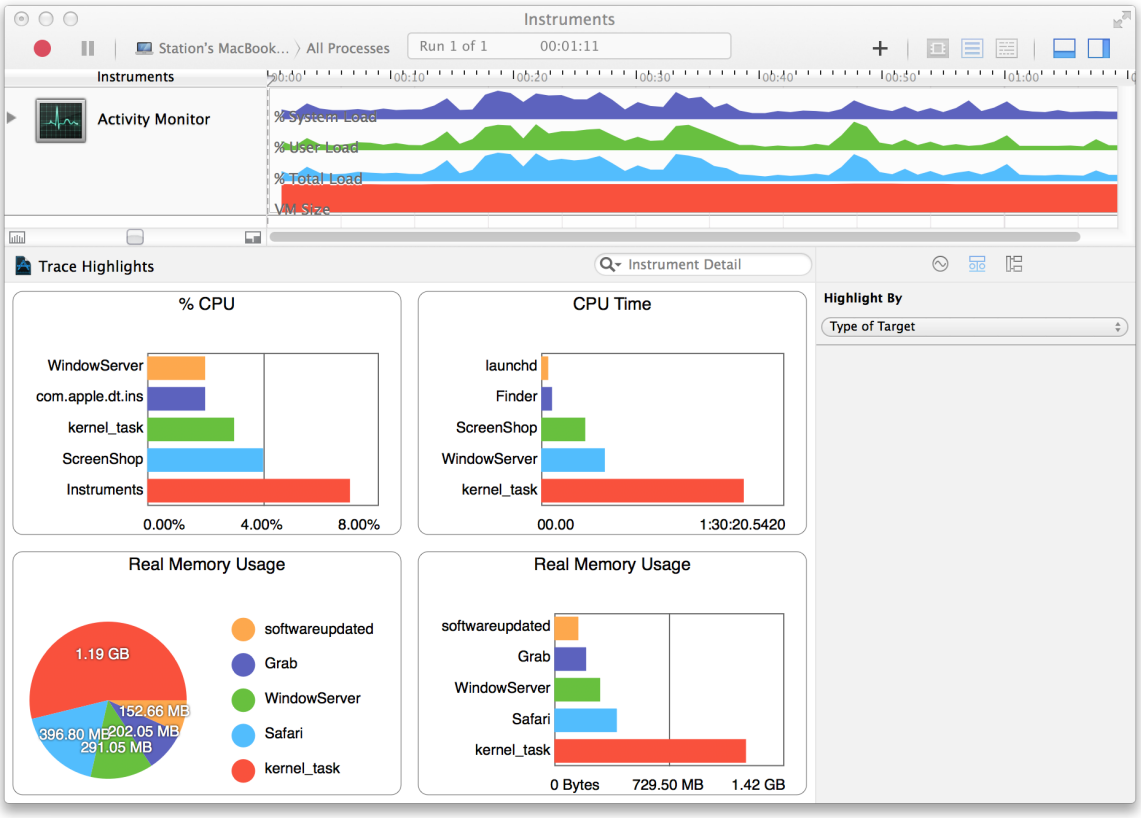
The Activity Monitor trace template monitors overall system activity and statistics, including CPU, memory, disk, and network. It consists of the Activity Monitor instrument only, although you can add additional instruments to a trace document you've created with the template, if you desire. You'll see later that the Activity Monitor is also used to monitor network activity on iOS devices.

The Activity Monitor instrument captures information about the load on the system measured against the virtual memory size. It can record information from a single process or from all processes running on the system. The Activity Monitor instrument provides you with four convenient charts for a quick, visual representation of the collected information. The two charts that specifically describe memory usage are:

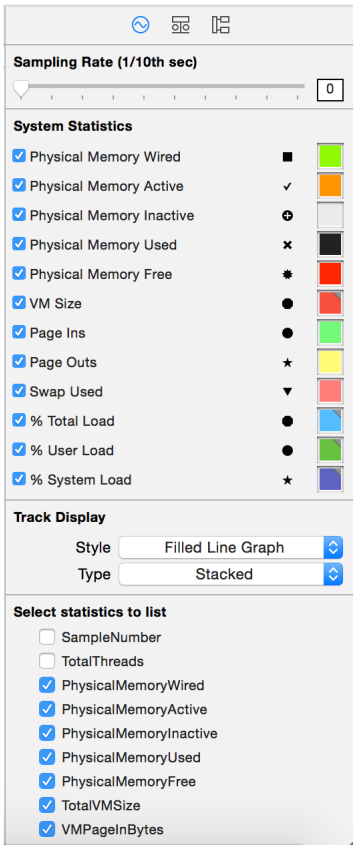
- **Real Memory Usage (bar graph).** Shows the top five real memory users in a bar graph.
- **Real Memory Usage (pie chart).** Shows the top five real memory users with the total memory used displayed.

Figure 7-1 shows the top five users of memory on the system.

Figure 7-1 Activity Monitor instrument with charts



The Record Settings area in the inspector sidebar includes a list of system statistics, which can be configured to appear in the track pane and graphically represent collected data. Select a statistic's checkbox to see it graphed in the track pane. Click the shape or the color well to change how a statistic appears in the track pane.



There are a number of statistics the Activity Monitor instrument supports, but the following ones are memory-specific:

- Physical Memory Wired
- Physical Memory Active
- Physical Memory Inactive
- Physical Memory Used
- Physical Memory Free
- Total VM Size
- VM Page In Bytes
- VM Page Out Bytes
- VM Swap Used

If one of the statistics above doesn't appear under System Statistics, locate it under "Select statistics to list" and click its checkbox to include it in the list.

Finding Abandoned Memory with the Allocations Trace Template

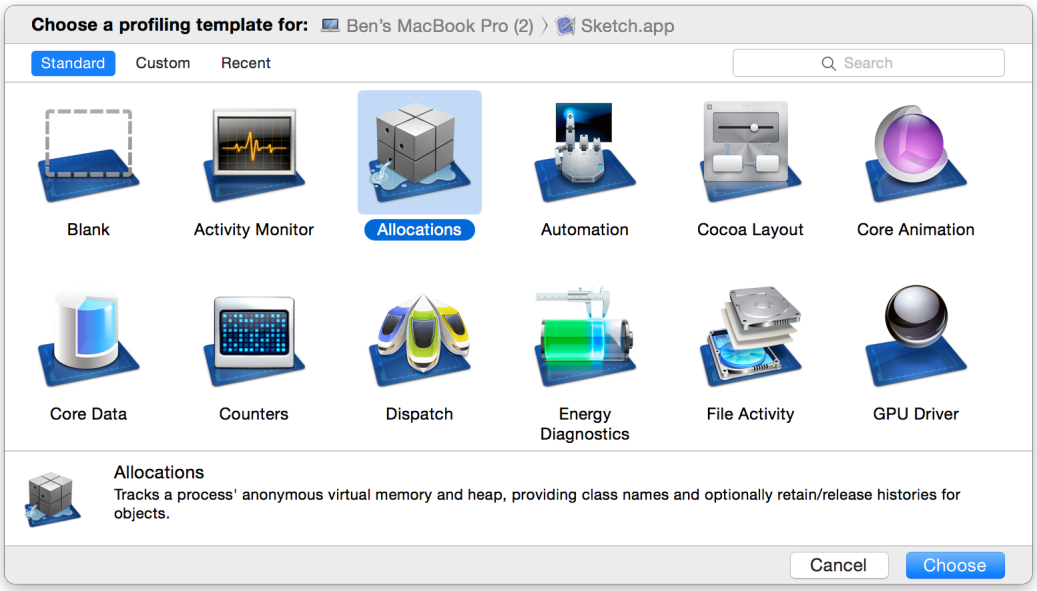
You can use the Allocations trace template and to identify abandoned memory in your app. Abandoned memory is different from leaked memory. Leaked memory is memory that you previously allocated but is no longer referenced anywhere. In other words, you no longer have any way to release it. Abandoned memory is memory that you allocated for some reason but simply don't need anymore. Perhaps your app contains some code for a feature you never fully implemented, or maybe you have a bug in your code that adds images to a cache after they've already been cached. These kinds of things can negatively impact your overall memory footprint by resulting in abandoned memory, and you should address them to make your app as efficienct as possible.

Because abandoned memory is still technically valid, just not useful, the Leaks instrument has no way to identify it. Therefore, you must perform some detective work to locate the problem. The Allocations trace template helps with this process by measuring heap memory usage and tracking allocations, including specific object allocations by class. It also records virtual memory statistics by region. It includes both the Allocations and VM Tracker instruments. For the purpose of locating abandoned memory, you'll use strictly the Allocations instrument.

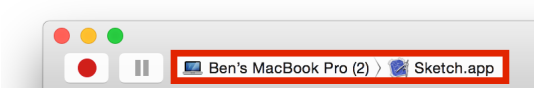
Use the Allocations template to ensure that the heap does not continue to grow while repeatedly performing the same set of operations. For example, ending and starting a new game, opening and closing a window, or setting and unsetting a preference are all operations that should conceptually return your app to a previous and stable memory state. Cycling through such operations many times should not result in unbounded heap growth. This process of repetition and analysis is known as *generational analysis*. A "generation" represents a set of allocations that occurred over a specified period of time. By repeating actions over multiple generations, you can analyze the results in order to identify allocation trends. When you find discrepencies in these trends, you can investigate further to determine whether memory is being abandoned, and if so, where. Then, you can fix it.

To find memory abandoned by your app

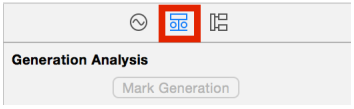
1. Open the Allocations trace template.



2. Choose your app from the Choose Target pop-up menu in the toolbar.

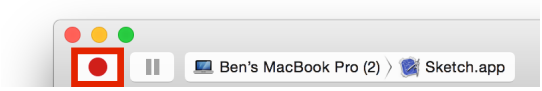


3. Click Display Settings in the inspector sidebar.



This gives you quick access to the Mark Generation button, which you'll use to flag different generations of actions as you profile your app.

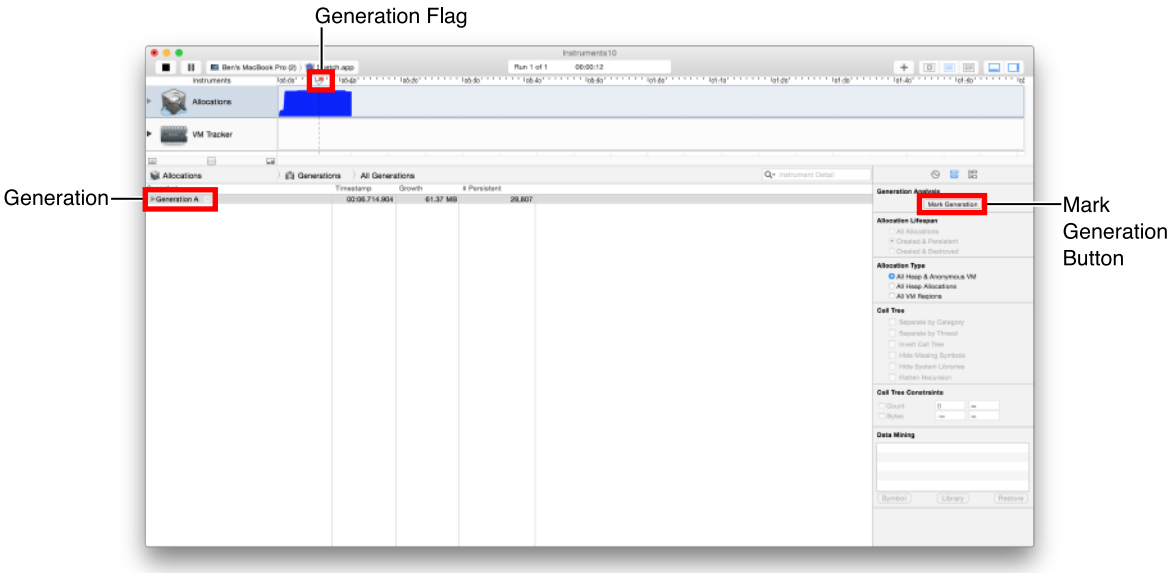
4. Click the Record button in the toolbar to begin profiling your app.



5. Perform a short sequence of repeatable actions in your app.

   In order to accurately generate trends, this should be a set of actions that starts and finishes in the same state.

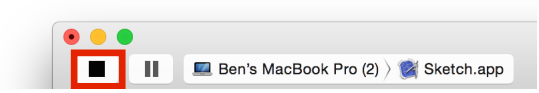
6. After each iteration of the repeatable actions, click the Mark Generation button in the inspector sidebar.



A flag appears in the track pane to identify the generation.

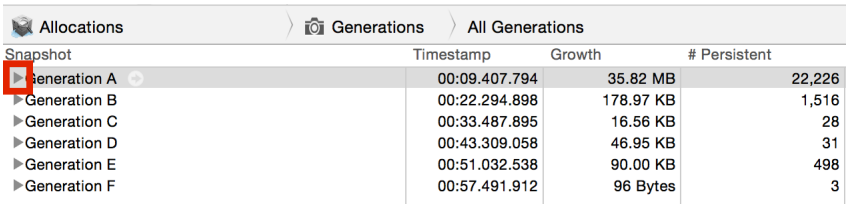
A list of generations you've marked is shown in the detail pane. Each generation includes a list of allocations that has occurred since the previous generation.

7. Repeat steps 5 and 6 several times, until you see whether or not the heap is growing without limit; then click the Stop button.



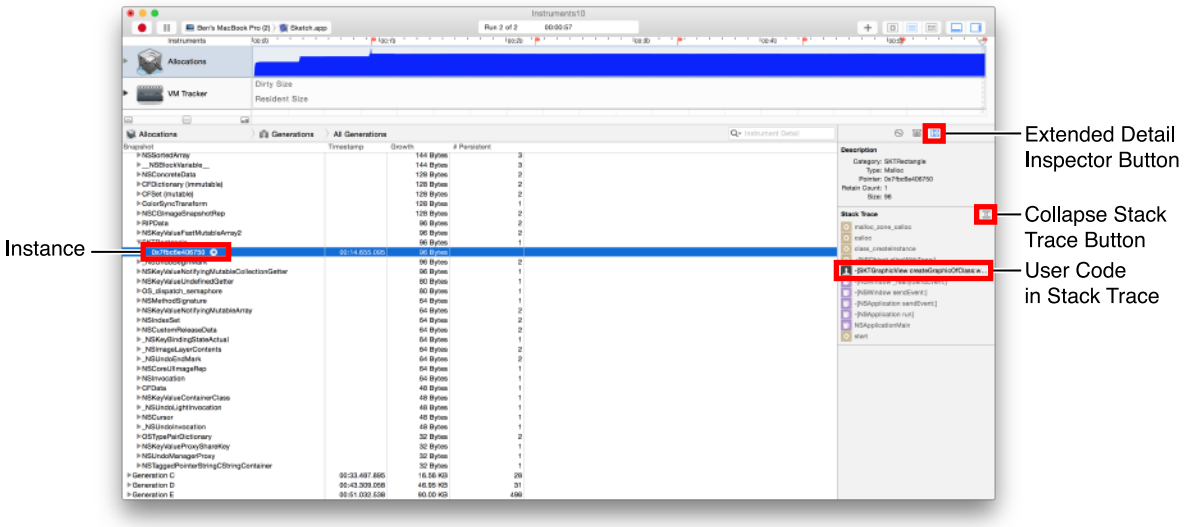
Note: Caches may be warming during the first few iterations and extra allocations may occur at this time. Therefore, it is important to create a few initial generations for the purpose of establishing a baseline. Then, you can create additional generations to use for true analysis. Generations can also be marked after you're done recording. Simply drag the inspection head in the track pane's timeline to desired location and click Mark Generation in the inspector sidebar.

8. In the detail pane, click the disclosure triangle to the left of a generation to display the objects it contains.



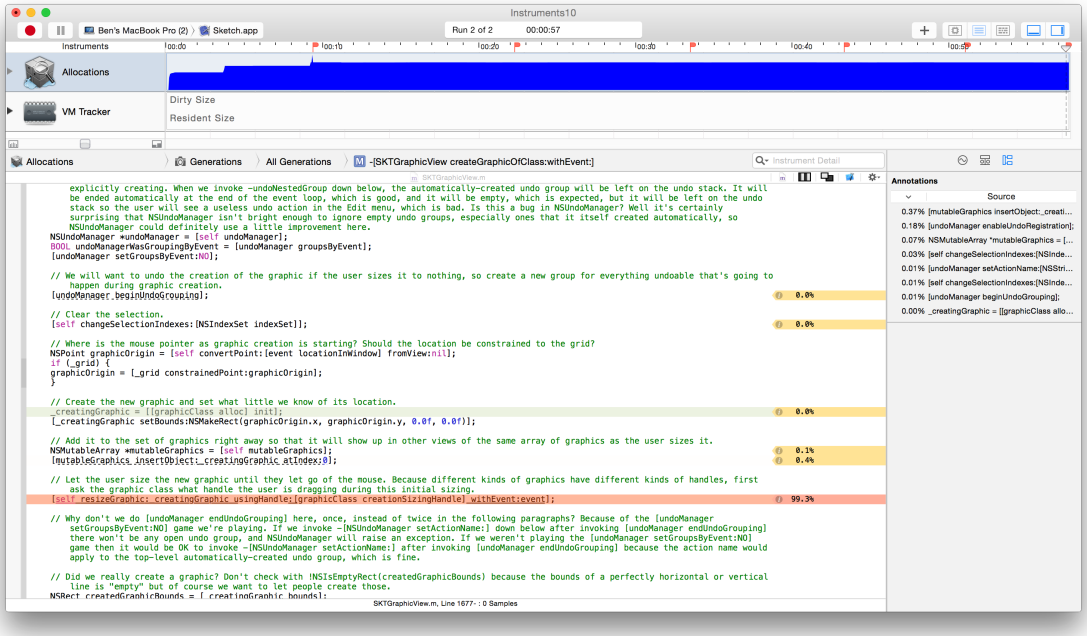
9. Look for objects that your app may be allocating, which are persisting. If you identify one, click the disclosure triangle to the left of it to display its instances.

10. Select an instance and click Extended Detail in the inspector sidebar to view the stack trace for that allocation.

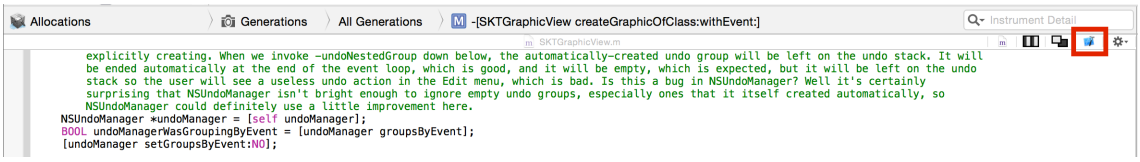


In the stack trace, your code is easily identifiable because it's colored black and preceded by a user icon. To make your code even easier to find, click the collapse button to hide system calls in the stack trace.

11. Double-click on a stack trace entry to display its corresponding code in the detail pane.



You can click the Xcode icon in the detail pane navigation bar to display the code in Xcode if you prefer.



12. Evaluate your code in order to determine whether the allocation is useful or not. If it's not, then it's abandoned memory and you should resolve it.

Note: In addition to helping you identify abandoned memory, generational analysis can also be useful for locating leaked and cached memory. As explained above, leaked memory (also detectable by using the Leaks instrument) is unreferenced by your application and cannot be freed or used again. Cached memory is memory that is still referenced by your application and might be used again for better performance.

Finding Leaks in Your App with the Leaks template

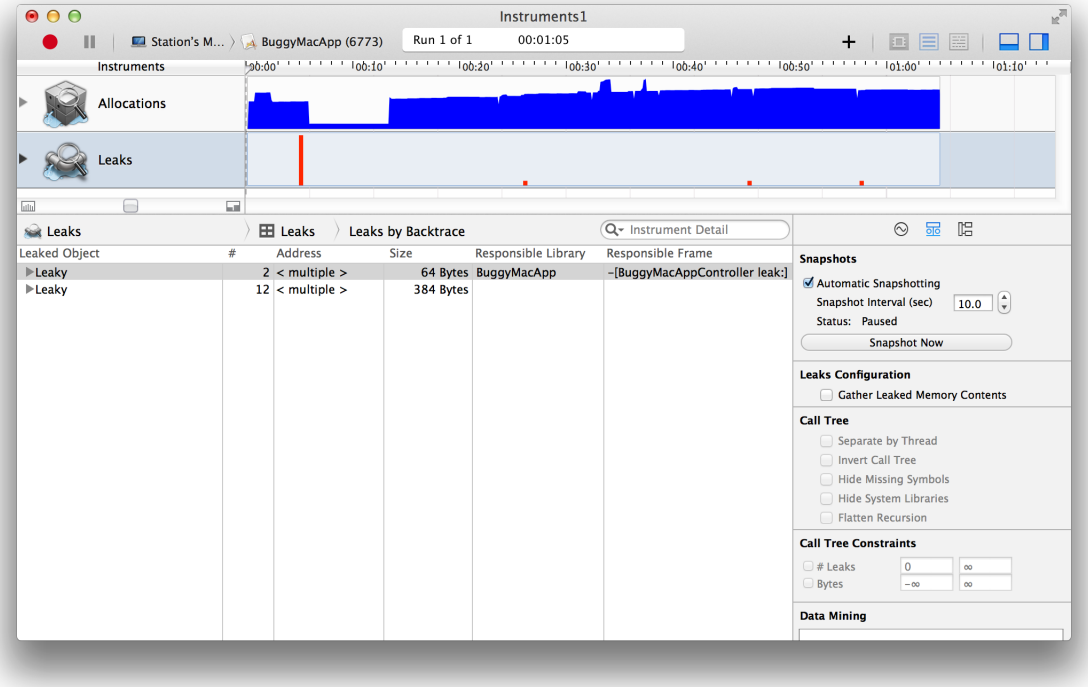
The Leaks trace template measures general memory usage, checks for leaked memory, and provides statistics on object allocations by class as well as memory address histories for all active allocations and leaked blocks. It consists of the Allocations and Leaks instruments.

Use the Leaks instrument to find objects in your app that are no longer referenced and reachable. The Leaks instrument reports these blocks of memory. Most of these leaks are objects and are reported with a class name. The others are reported as Malloc-*size*.

To locate leaking memory

1. Open the Leaks template.
2. From the Choose Target pop-up menu in the toolbar, choose your app.
3. Click the Record button.
4. Exercise your app in order to execute code, and click Stop when leaks are displayed.
5. Click any leaked object that is identified in the Detail pane.
6. Within the Extended Detail inspector, double-click an instruction from your code.

7. Click the Xcode icon in the Detail pane to open that code in Xcode.



Note: If a leak isn't an object, you may be calling an API that assumes ownership of a `malloc`-created memory block for which you are missing a corresponding call to `free()`.

If the leaked memory is an Objective-C object, then pressing the focus arrow next to the address of an individual object will show the retain/release history of the object with corresponding stack traces for each event.

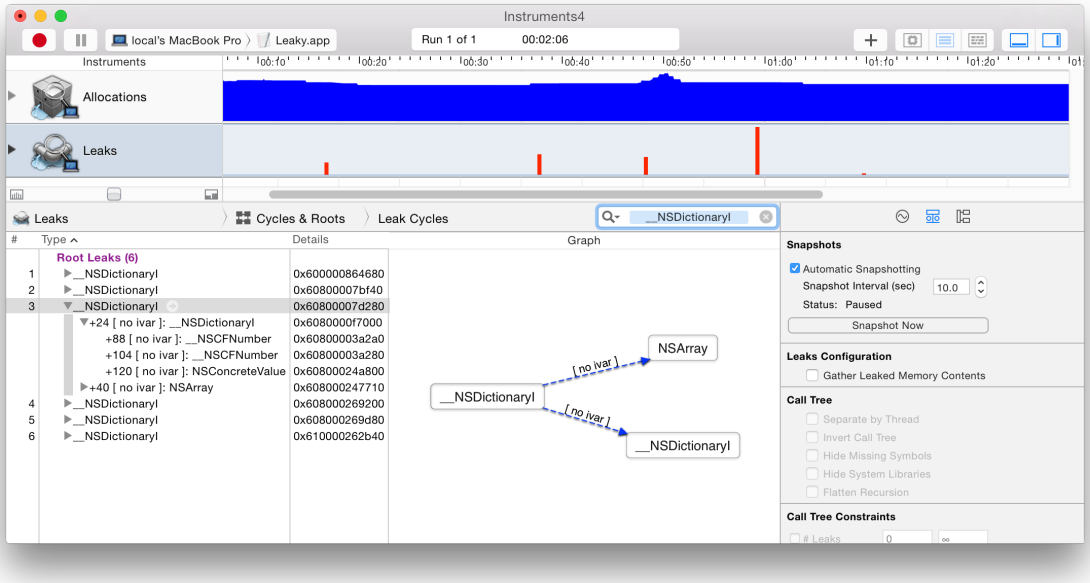
Non-object leaks are the result of calling `malloc()` without a corresponding `free()` and thus only the stack trace of the allocation event is presented.

The Leaks by Backtrace view aggregates all of the leaked blocks by their allocation point — a single mistake in source code can result in multiple runtime leaks as the code is executed repeatedly.

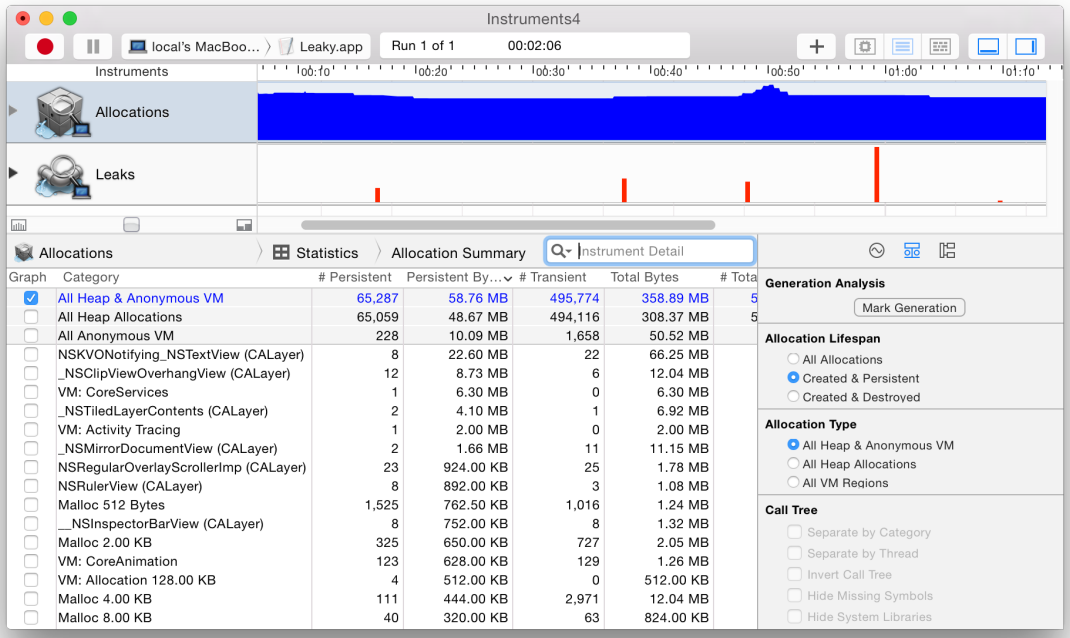
After opening Xcode to see the piece of code that is creating the leak, the cause of the leak may still be unclear. The Leaks instrument allows you to see the cycle that is creating the leak in the Cycles & Roots option in the detail pane.The Cycles & Roots view is important when writing apps using Automatic Reference Counting (ARC) as it gives a graph-level view of leaked objects, aiding in the diagnosis of retain cycles. Frequently the solution to these problems will be marking an up-reference in the object graph with the `weak` keyword.

To see the cycle graph of a leak

1. Select the Leaks instrument.
2. Select Cycles & Roots in the detail pane.
3. Select the leak whose graph you want to see.



The Allocations instrument also shows VM regions by type, including CALayer regions identified by their delegate object classes. This lets you see allocation backtraces for those regions, unlike in the VM Tracker instrument. This can be useful for optimizing space in graphics-heavy apps since often ObjC objects are referencing custom graphics VM regions.



Tip: App instability after a fix may result if the fix is incorrect or if another part of your code still relies on the eliminated bug.

Eradicating Zombies with the Zombies Trace Template

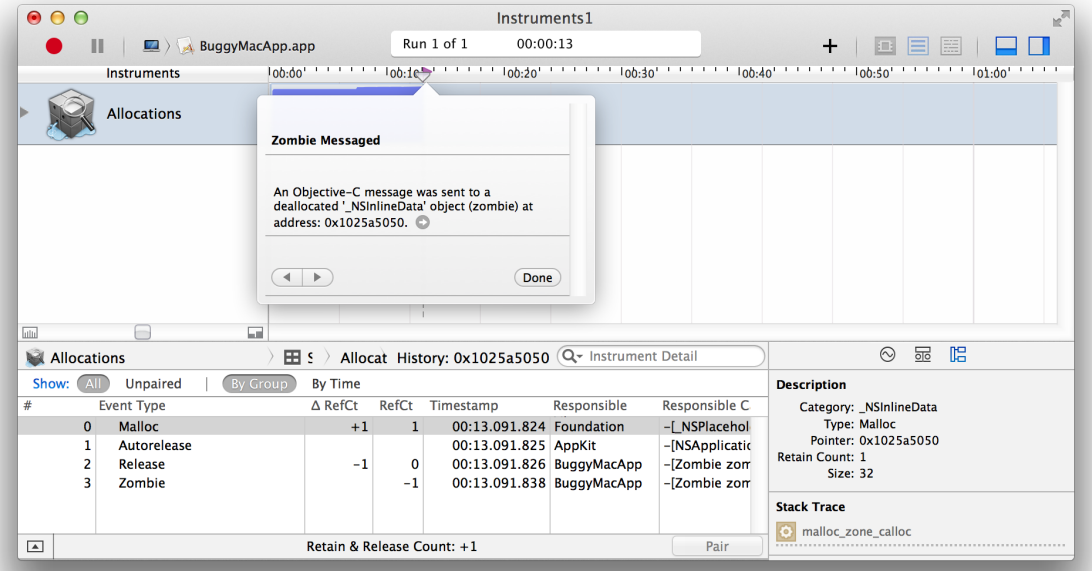
The Zombies trace template measures general memory usage while focusing on the detection of overreleased, "zombie" objects. It uses the Allocations instrument to display statistics on object allocations by class as well as memory address histories for all active allocations. Note that, because the Zombies trace template uses a "debug" mode, many other values displayed by the Allocations instrument will not be meaningful because the memory is not actually deallocated.

The Zombies trace template sets the environment variable `NSZombieEnabled` to `true` which directs the compiler to substitute an object of type `NSZombie` for objects that are released to a reference count of zero. Then, when a zombie is messaged, the app crashes, recording stops, and a Zombie Messaged dialog appears. Clicking the focus button to the right of the message in the Zombie Detected dialog displays the complete

memory history of the overreleased object. This allows you to examine the retain and release history of the problematic zombie object, to try to determine if there was an extra release (or autorelease) that should not have been there, or maybe a missing retain. Note that this memory history is the only part of the Allocations Instrument that is meaningful when using the Zombies trace template.

To find zombies in your code

1. Open the Zombies template.
2. Choose your app from the Choose Target pop-up menu.
3. Click the Record button and exercise your app.
4. When a Zombie Messaged dialog appears, click the focus button to the right of the message text in the dialog.
5. Open the Extended Detail inspector in the sidebar pane, and double-click the Zombie entry in the Event Type column of the object history table.
6. Double-click the responsible caller in the Stack Trace section of the Extended Detail inspector to display the responsible code.



When an `NSZombie` object is messaged, a Zombie Messaged dialog appears. Clicking the focus button to the right of the message in the Zombie Messaged dialog displays the complete memory history of the overreleased object.

Note that the Zombie Messaged dialog has a point at the top, aligned with a purple triangle in the timeline. If the Zombie Messaged dialog is no longer visible and you want to see it again, click the purple triangle.

> **Tip:** The Zombies template causes memory growth because it changes your environment so that the zombies are never deallocated. Therefore, for iOS apps, use the Zombies template with the iOS Simulator rather than on the device itself. For the same reason, don't use the Zombies template concurrently with the Leaks instrument.

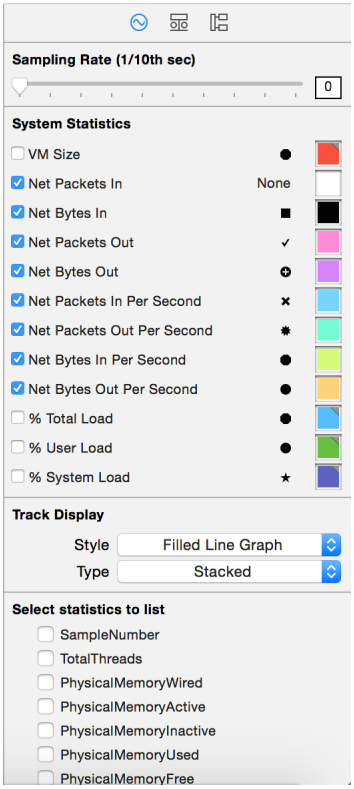
Measuring I/O Activity in iOS Devices

Apps can be complicated programs with a lot of information being passed between the device and the user. The I/O Activity trace template in Instruments help you see what your app is doing and where it is sending and receiving information. This chapter shows you how to use these trace templates and monitor your app's activity.

Following Network Usage Through the Activity Monitor Trace Template

The Activity Monitor trace template monitors overall system activity and statistics, including CPU, memory, disk, and network. It consists of the Activity Monitor instrument only, although you can add additional instruments to a trace document you've created with the template, if you desire.

By default, the Activity Monitor template isn't set up to display network activity. Therefore, you need to enable the desired network-related statistics in the Record Settings area in the inspector sidebar for the Activity Monitor instrument to see which processes are sending and receiving information.



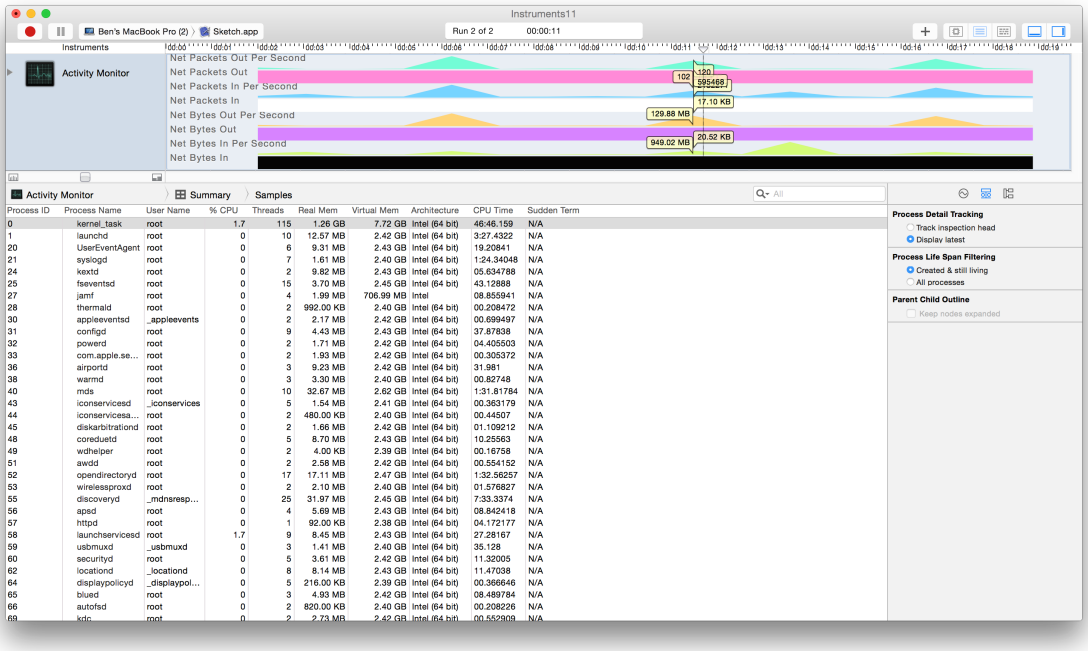
There are a number of statistics the Activity Monitor instrument supports, but the following ones are network-specific:

- Net Packets In
- Net Bytes In
- Net Packets Out
- Net Bytes Out
- Net Packets In Per Second
- Net Packets Out Per Second
- Net Bytes In Per Second
- Net Bytes Out Per Second

If one of the statistics above doesn't appear under System Statistics in the Record Settings inspector, locate it under "Select statistics to list" and click its checkbox to include it in the list.

Once you have gathered network activity for your app, examine it carefully to pinpoint areas where your app is sending out excessive amounts of information and therefore tying up valuable device resources. When you minimize the amount of information sent and received, you can benefit from increased performance and response times in your app.

Figure 8-1 Activity Monitor instrument tracing network packets



Analyzing iOS Network Connections with the Network Trace Template

The Network template analyzes how your iOS apps are using TCP/IP and UDP/IP connections. The Network trace template consists of the Connections instrument.

To view network connections used by your app

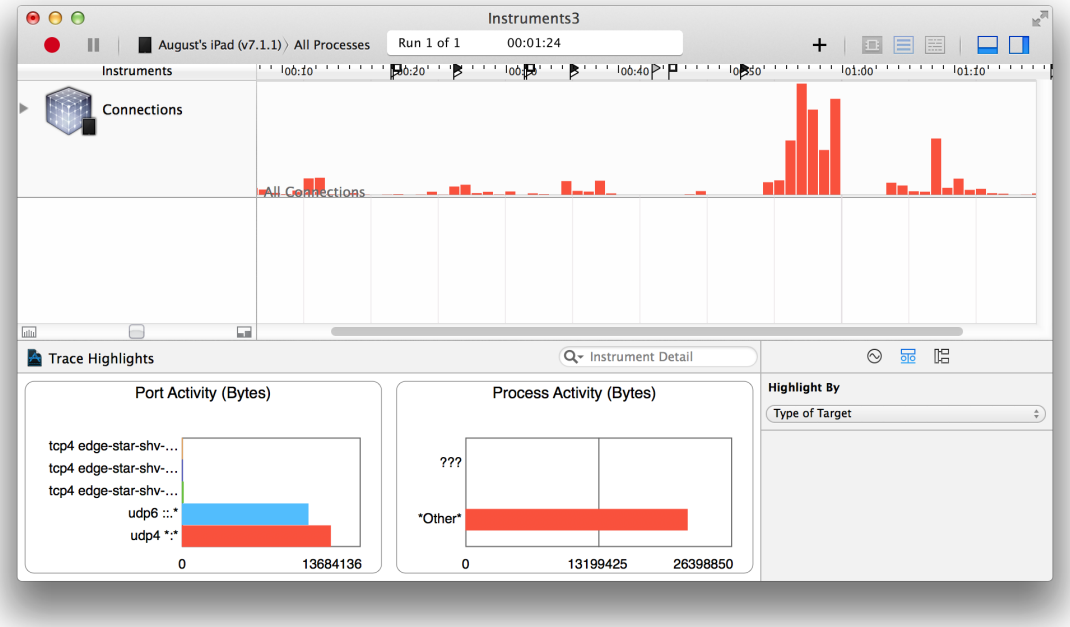
1. Connect your iOS device.

   You can use a physical or wireless connection. See To enable an iOS device for wireless profiling (page 36).

2. Choose a target from the Target pop-up menu in the trace document toolbar.
3. Click Record and exercise your app.
4. Click Stop.

Selecting Trace Highlights in the detail pane provides two bar graphs: The first bar graph lists the top five active ports and the amount of information that has traveled through them. The second one lists the quantity of bytes used by other processes.

Figure 8-2 Viewing network connections



Switching to the Connection view in the detail pane shows the collected information in column form. By default, the track pane displays a graph of all connections. However, in the Connection view area in the detail pane, you can click checkboxes to graphically organize the data by individual processes, providing a means for comparison.

Tip: You may find it useful, in conjunction with the Connections instrument, to use the Network Activity instrument to measure the number of packets that are sent and received by your app.

Measuring Graphics Performance on Your iOS Device

Extensive use of graphics in your iOS app can make your app stand out from your competitor. But unless you use graphics resources responsibly, your app will slow down and look mediocre no matter how good the images you are trying to render.

Use the trace templates outlined in this section to profile your app. Ensure that the frame rate is high enough and that your graphics don't impede your app's performance.

Measuring Core Animation Graphics Performance in iOS with the Core Animation trace template

Instruments uses the Core Animation instrument to measure an app's graphical performance on an iOS device. The Core Animation trace template provides a quick and lightweight starting point for using this instrument, allowing you to measuring the number of frames per second rendered by your app. It allows you to quickly see where your app renders fewer than expected frames. By correlating what you were doing at the time the sample was taken, you can then identify areas of your code that need to be improved.

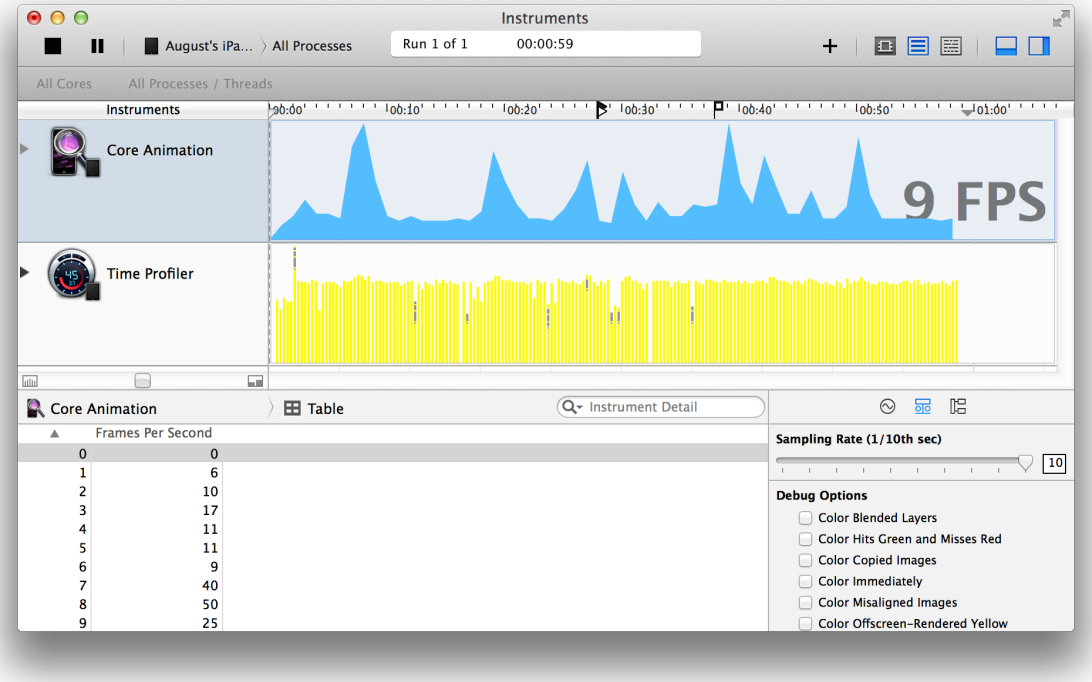
> **Note:** The Core Animation instrument is not intended to be used to measure OpenGL ES performance.

Correlate your interactions with your app with the results displayed in Instruments

In Figure 9-1, you can see spikes where the frame rate of the app becomes appreciably better. Without knowing what was happening with the device during these spikes, it would be natural for you to want to duplicate these higher frame rates throughout the app. However, these spikes were caused by orientation changes when

the device was changed between landscape and normal orientation. Without knowing that an orientation change by the device was performed, you might spend time trying to find what caused the performance increase and replicating it throughout your app.

Figure 9-1 Core Animation trace template showing frame rate spikes



Debugging options

Core Animation contains several useful debugging options, which can be enabled/disabled in the Display Settings area in the inspector sidebar. You do not need to be running a trace in order to see these options working on your iOS device.

- **Color Blended Layers.** Shows blended view layers. Multiple view layers that are drawn on top of each other with blending enabled are highlighted in red. Reducing the amount of red in your app when this option is selected can dramatically improve your apps performance. Blended view layers are often the cause for slow table scrolling.
- **Color Hits Green and Misses Red.** Marks views in green or red. A view that is able to use a cached rasterization is marked in green.
- **Color Copied Images.** Shows images that are copied by Core Animation in blue.
- **Color Immediately.** When selected, removes the 10 ms delay when performing color-flush operations.

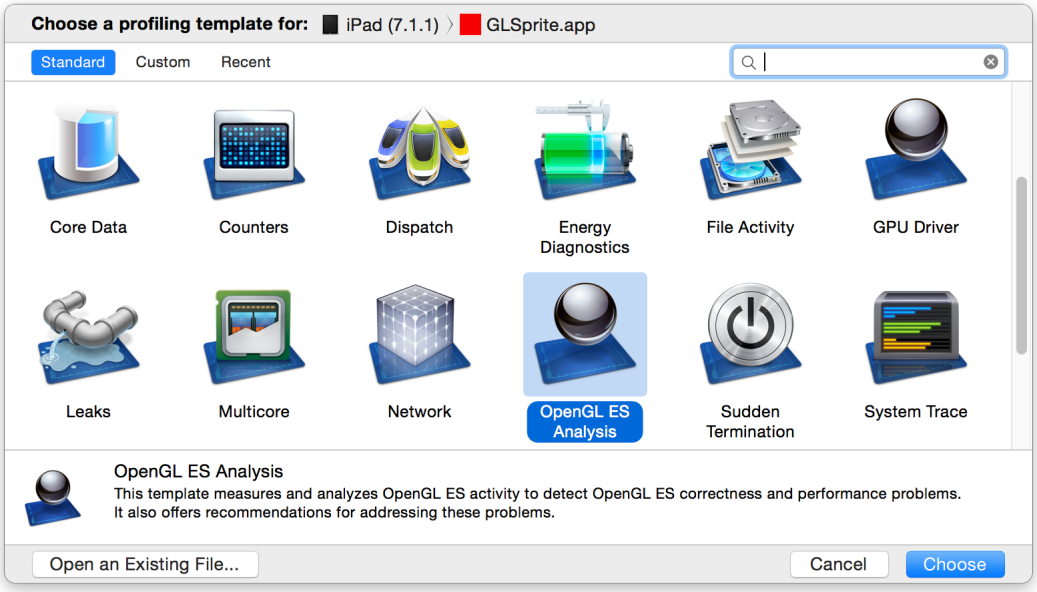
- **Color Misaligned Images.** Places a magenta overlay over images where the source pixels are not aligned to the destination pixels.
- **Color Offscreen-Rendered Yellow.** Places a yellow overlay over content that is rendered offscreen.
- **Color OpenGL Fast Path Blue.** Places a blue overlay over content that is detached from the compositor.
- **Flash Updated Regions.** Colors regions on your iOS device in yellow when that region is updated by the graphics processor.

Measuring OpenGL Activity in iOS with the OpenGL ES Analysis Trace Template

The OpenGL ES Analysis template measures and analyzes OpenGL ES activity in order to detect OpenGL ES correctness and performance problems. It also offers you recommendations for addressing found problems. It consists of the OpenGL ES Analyzer and the GPU Driver instruments.

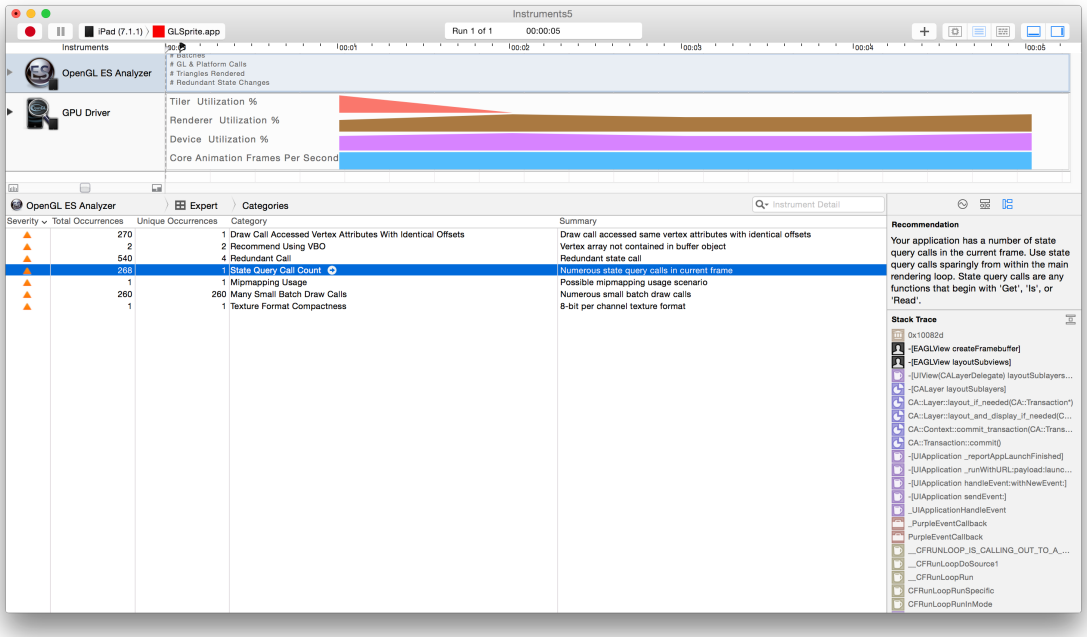
To get OpenGL ES Analyzer to make suggestions for your app

1. Create a new trace document in Instruments using the OpenGL ES Analysis template. At the top of the template selection window, choose the device and app you want to analyze. Or, you can choose a target device and app from the trace document's toolbar, once it is created.



2. Click the Record button, and exercise your OpenGL ES graphics code in your app.

3. After a few seconds of measurement, click the Stop button and wait for issues to stop accumulating in the detail pane.

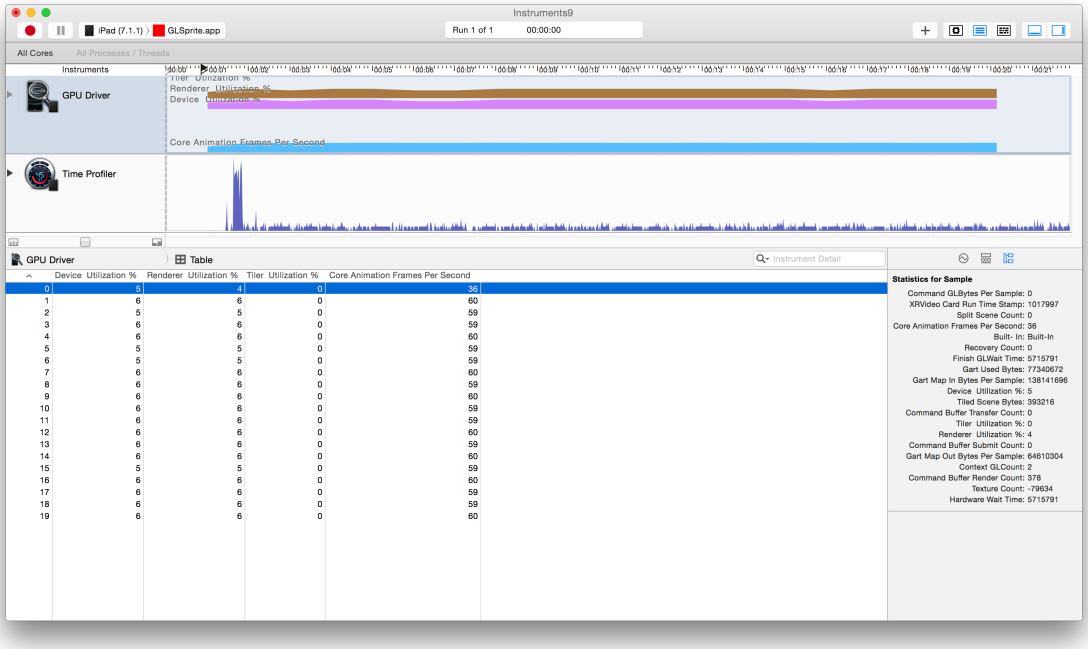


Errors are listed in the detail pane sorted by their severity. Red squares are used for “most severe” problems, and orange triangles are used for “less severe” problems. A recommendation and the stack trace are displayed in the Extended Detail inspector pane for the issue selected in the detail pane.

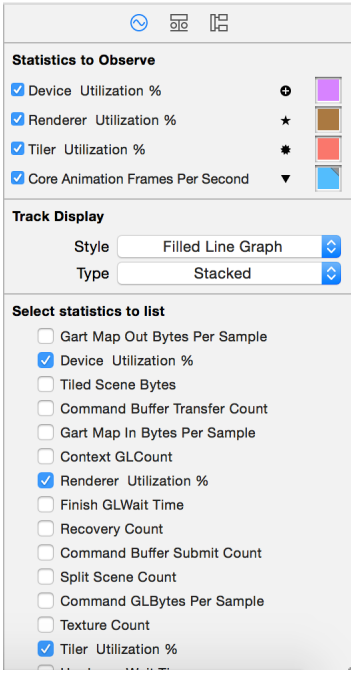
Finding Bottlenecks in an iOS app with the GPU Driver Trace Template

The GPU Driver trace template is also used to measure app performance and provides you with more information than just the number of frames per second that your app renders. The detail pane displays all of the gathered information for a specific sample. Select a row of data to view its statistics in the Extended Detail inspector.

Figure 9-2 Detailed information for a GPU Driver sample



The individual statistics displayed in the track pane can be enabled/disabled by adjusting the GPU Driver instrument's Record Settings in the inspector sidebar.



Bottlenecks for an OpenGL app often come in two forms, a GPU bottleneck or a CPU bottleneck. GPU bottlenecks occur when the GPU forces the CPU to wait for information as it has to much information to process. CPU bottlenecks often occur when the GPU has to wait for information from the CPU before it can process it. CPU bottlenecks can often be fixed by changing the underlying logic of your app to create a better overall flow of information to the GPU. The following shows a list of bottlenecks and common symptoms that point to the bottleneck:

- **Geometry limited.** See whether Tiler Utilization is high. If it is, then look into your vertex shader processes.
- **Pixel limited.** See whether Rendered Utilization is high. If it is, then look into your fragment shader processes.
- **CPU limited.** See whether Tiler and Rendered Utilization are low. If both of these are low, the performance bottleneck may not be in your OpenGL code and you should look into the your code's overall logic.

Analyzing CPU Usage in Your App

Ensuring effective use of all available resources is important when writing code for your app. One of the most important resources is the CPU. Effective use of the CPU allows your app to run faster and more effectively. Even though you will be writing your app for a particular platform, keep in mind that even the same general platform type can have different CPU capabilities. The CPU trace templates provide you with the means to conduct testing under a variety of potential conditions, allowing you to identify how well your app uses multiple cores, how much energy you are using, and other resource measurements.

Looking for Bottlenecks with Performance Monitor Counters

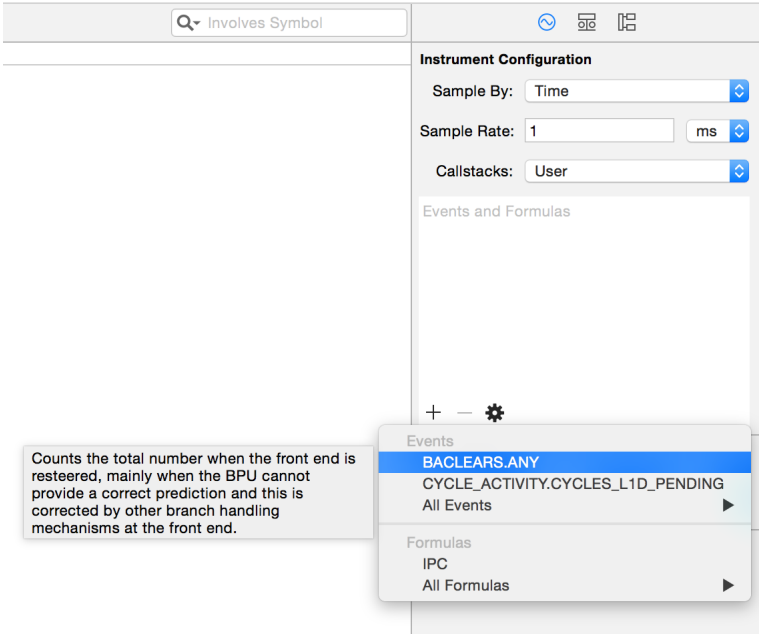
Performance monitor counters (PMCs) are hardware registers that measure events occurring in the processor. They can be used to help find bottlenecks in your app by identifying an excessive amount of events of a particular type. For example, a high number of conditional branch instructions may indicate a section of logic that, if rearranged, might lower the number of branches required. Even though PMC events can bring these issues to light, it is up to you to match them to your code and decide how they will help you improve your app's performance.

In Instruments, you track PMC events using the Counters instrument.

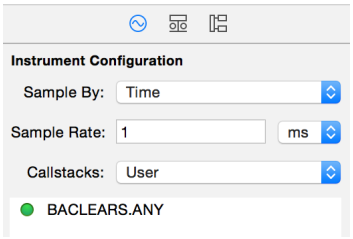
To track PMC events in the Counters instrument

1. Open the Counters trace template.
2. Choose an app from the target menu in the toolbar.

3. In the Record Settings inspector, click the plus button (+) in the Events and Formulas table.

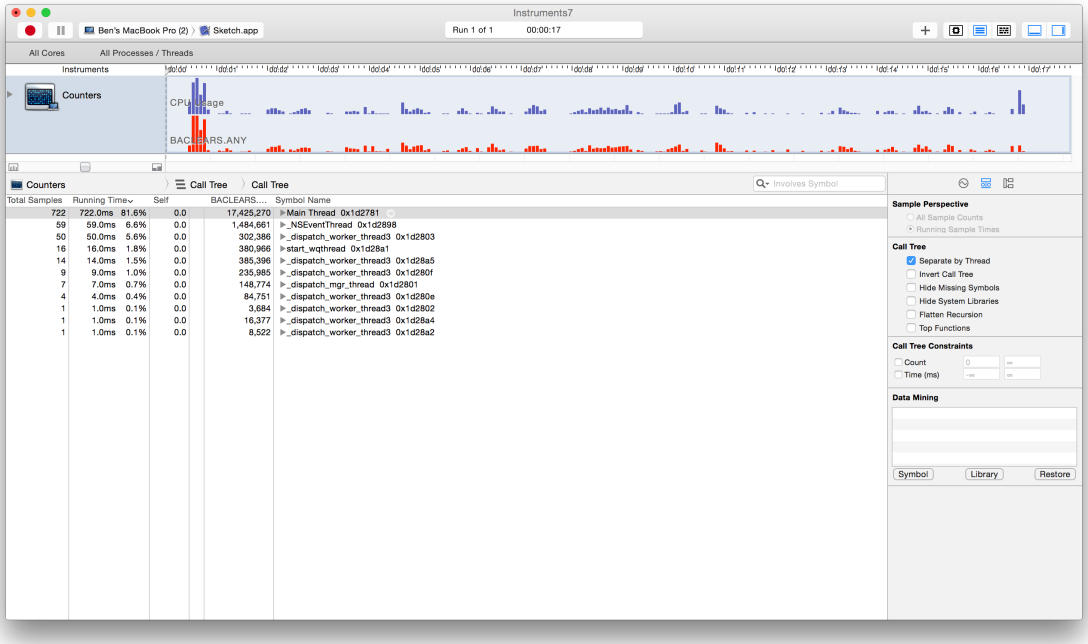


4. Select the event you'd like to count.



5. Repeat steps 3 and 4 to add more events, if desired.
6. Click Record and exercise your app.

7. Click Stop.



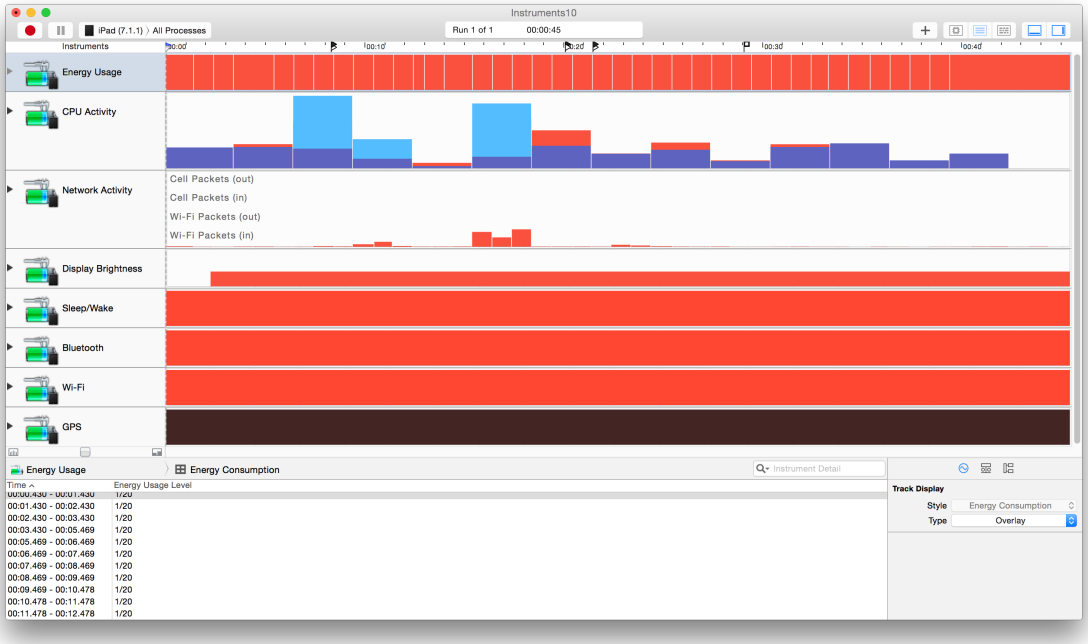
Important: The number of PMC events that can be tracked is hardware dependent. Trying to track too many events at one time can cause an error. Experiment with your setup to determine the number of events that can be successfully tracked at one time.

Tip: If you plan on recording the same PMC events frequently, save them in a trace template. Otherwise, they will be lost when you close the document. For information on saving a trace template, see Saving an Instruments Trace Template (page 56).

Saving Energy with the Energy Diagnostics Trace Template

The Energy Diagnostics trace template provides diagnostics regarding energy usage, as well as basic on/off states of major device components. This template consists of the Energy Usage, CPU Activity, Network Activity, Display Brightness, Sleep/Wake, Bluetooth, WiFi, and GPS instruments.

Figure 10-1 The Energy Diagnostics template after it has collected data from an iOS device



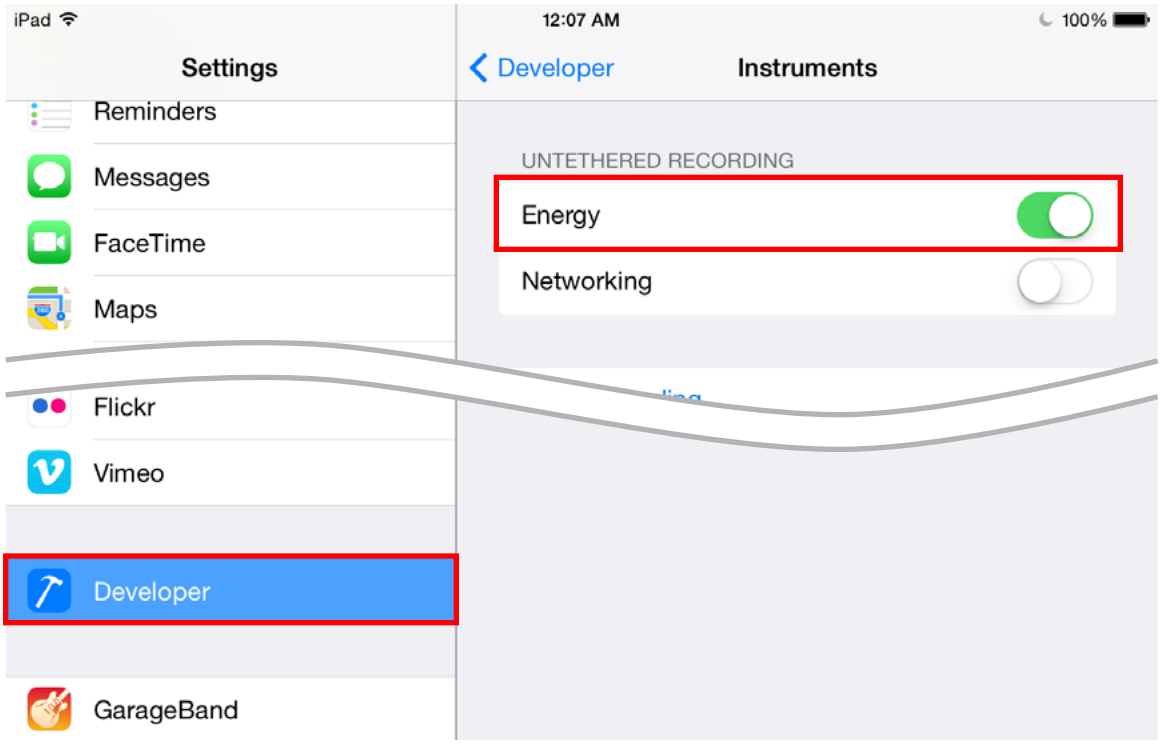
With Energy Diagnostics Logging on, your iOS device records energy-related data unobtrusively while the device is used. Because logging is efficient, you can log all day. Logging continues while the iOS device is in sleep mode, but if the device battery runs dry or the iOS Device is powered off, the log data is lost.

A Developer setting appears in the Settings app only after your device is provisioned for development. The setting disappears after the device has been rebooted. Restore the setting by connecting the device to Xcode or Instruments.

After sufficient energy usage events have been logged, you can analyze them by importing the log data from the phone to the Xcode Instruments Energy Diagnostics template. Look for areas of high energy usage and see whether you can reduce energy usage in these areas.

To track energy usage on an iOS device

1. Go into Settings > Developer and enable developer logging on the iOS device where you want to capture data.



2. Exercise your app like a user would.
3. After capturing the data, turn off developer logging.

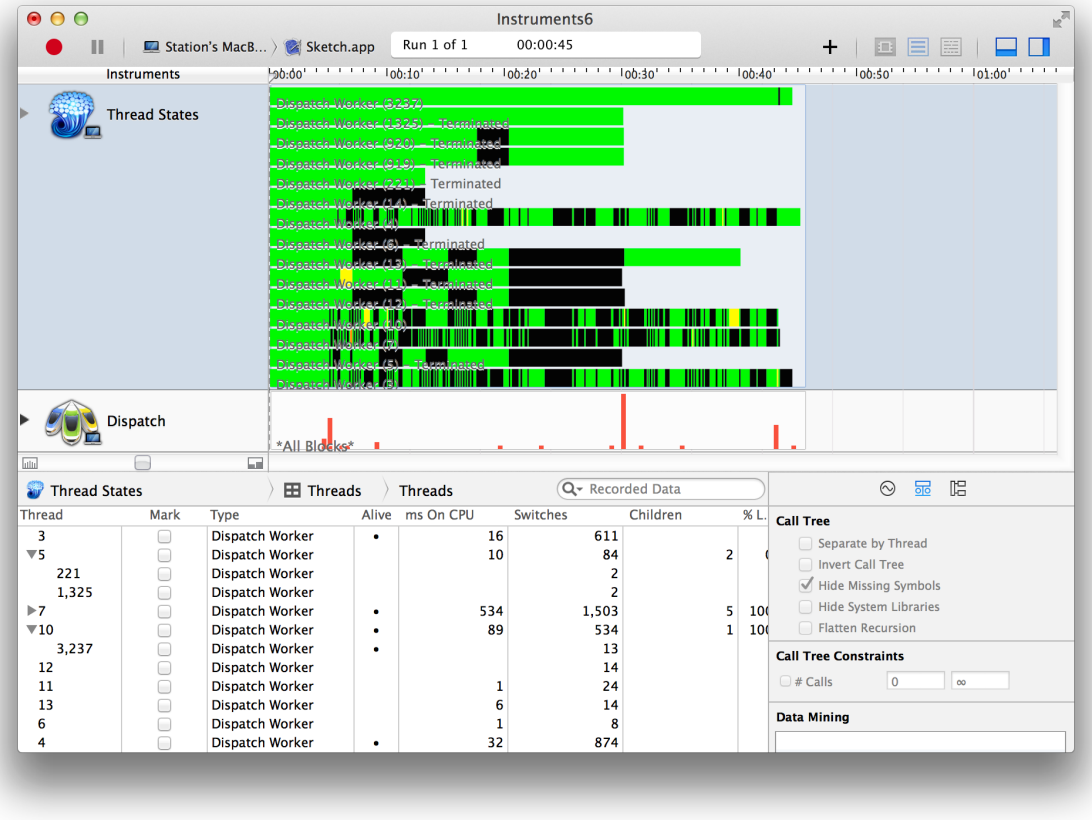
Minimize the amount of energy your app uses by ensuring that you turn off any device radios that you don't actively need. You can verify that you have turned off a particular radio by using the Energy Diagnostics trace template. In the track pane, each radio is depicted with red to designate that it is on or with black to designate that it is turned off. In Figure 10-1, the GPS radio track is disabled and therefore its track appears in black.

Examining Thread Usage with the Multicore Trace Template

The Multicore trace template analyzes your multicore performance, including thread state, dispatch queues, and block usage. It consists of the Thread States and the Dispatch instruments.

The Thread States instrument provides you with a graphical representation of each thread's state at a particular time in the run. Each state is color-coded to help you identify what each thread is doing in conjunction with all of the other threads. Threads that go through multiple state changes are easily identifiable through the changing colors in the track pane. Figure 10-2 shows fifteen threads being tracked.

Figure 10-2 Thread activity displayed by the Multicore trace template



To view thread usage in your app

1. Create a new trace document using the Multicore template.
2. Choose your app from the target menu in the toolbar.
3. Run your app.
4. Select the Thread States instrument in the Instruments pane.
5. Choose the threads to examine by selecting the checkbox in the Mark column in the detail pane.

The following thread states are captured in the track pane by the Thread States instrument. Color notations are the default colors, but they can be changed by you.

- **Unknown / At termination (Gray).** Instruments is unable to determine the state of the thread or it is being terminated.
- **Waiting (Yellow).** The thread is waiting for another thread to perform a particular action.
- **Suspended (Dark Blue).** The thread has been put into a suspended state and will not continue until it is specifically told to resume running.
- **Requested to suspend (Light Blue).** The thread has sent out a request to be put into a suspended state.
- **Running (Green).** The thread is running.
- **On run queue (Black).** The thread is in the queue to be run. It will run after a CPU becomes available.
- **Waiting uninterruptedly (uninterruptibly) (Orange).** The thread is waiting for another thread to perform a particular action and can not be interrupted during the wait.
- **Idling processor (White).** The thread is active on a processor, but is not performing any actions.

Along with the Thread States instrument, the Multicore template contains the Dispatch instrument. Use the Dispatch instrument to see when your dispatch queues are executed. You can see how long the dispatched thread lasts and how many blocks are used.

The Multicore trace template displays thread interaction throughout your app. However, you are not able to see which cores are being used. To see core usage by your app, see Delving into Core Usage with the Time Profiler Trace Template (page 91).

Delving into Core Usage with the Time Profiler Trace Template

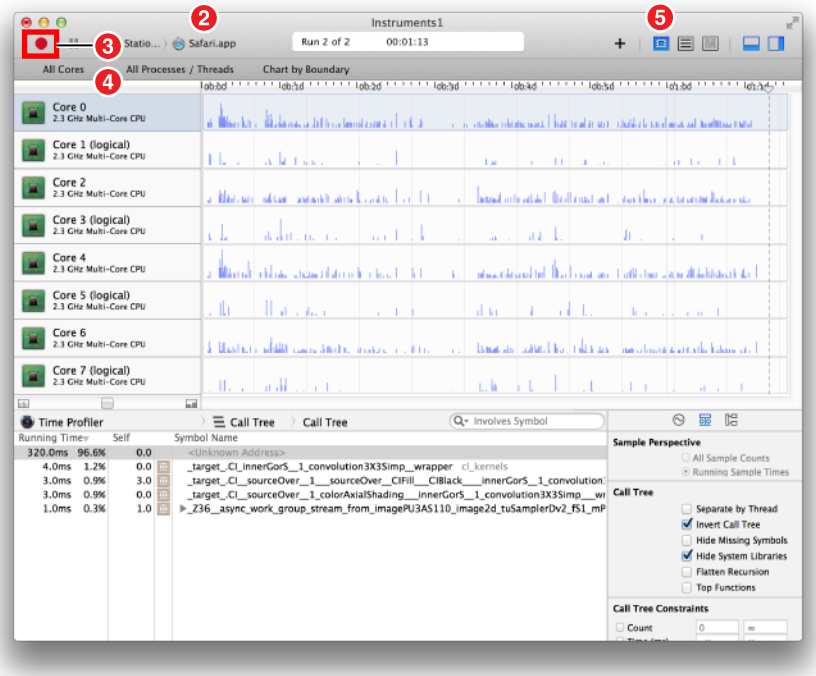
The Time Profiler trace template performs low-overhead, time-based sampling of processes running on the system's CPUs. It consists of the Time Profiler instrument only.

The CPU strategy in the Time Profiler instrument shows how well your app utilizes multiple cores. Selecting the CPU strategy in a trace document configures the track pane to display time on the x-axis and processor cores on the y-axis. Use the CPU strategy usage view to compare core usage over given time periods. Effective core concurrency improves an app's performance. Areas of heavy usage for a single core while other cores remain quiet can depict areas needing greater optimization.

To view individual core usage

1. Open the Time Profiler trace template.
2. Choose your app from the Choose Target pop-up menu in the trace document's toolbar.
3. Click the Record button.
4. Exercise your app to execute code, then click the Stop button to capture and analyze data.

5. Click the CPU strategy button.
6. Choose whether you want to graphically display the chart by boundary, category, or usage in the lower section of the toolbar.
7. Look for unbalanced core usage.



Ensure that your app is using multiple cores simultaneously by zooming in on the track pane. One or two threads that jump between cores very quickly can make it look like multiple cores are in use at the same time when in reality, only one core is in use at any one time.

Automating UI Testing

When you automate tests of UI interactions, you free critical staff and resources for other work. In this way you maximize productivity, minimize procedural errors, and shorten the amount of time needed to develop product updates.

You can use the Automation instrument to automate user interface tests in your iOS app through test scripts that you write. These scripts run outside of your app and simulate user interaction by calling the UI Automation API, a JavaScript programming interface that specifies actions to be performed in your app as it runs in the simulator or on a connected device. Your test scripts return log information to the host computer about the actions performed. You can even integrate the Automation instrument with other instruments to perform sophisticated tests such as tracking down memory leaks and isolating causes of performance problems.

This chapter describes how you use the Automation template in Instruments to execute scripts. The Automation trace template executes a script which simulates UI interaction for an iOS app launched from Instruments. It consists of the Automation instrument only.

This chapter also explains how to integrate your scripts with the UI Automation programming interface in order to verify that your app can do the following:

- Access its UI element hierarchy
- Add timing flexibility by having timeout periods
- Log and verify the information returned by Instruments
- Handle alerts properly
- Handle changes in device orientation gracefully
- Handle multitasking

The Automation instrument provides powerful features, including:

- Script editing with a built-in script editor
- Capturing (recording) user interface actions for use in automation scripts
- Running a test script from an Xcode project
- Powerful API features, including the ability to simulate a device location change and to execute a task from the Automation instrument on the host

As you work through this chapter, look for more detailed information about each class in *UI Automation JavaScript Reference for iOS*. For an overview of UI Automation with JavaScript, see *JavaScript for Automation Release Notes*. For some sample automation projects, see *JavaScript for Automation WWDC 2014 Demos*.

Note: The Automation instrument works only with apps that have been code signed with a development provisioning profile when they are built in Xcode. Apps signed with a distribution provisioning profile cannot be controlled with the UI Automation programming interface. However, because scripts run outside your app, the app version you are testing can be the same one you submit to the App Store, as long as you rebuild it with the distribution profile.

Important: Simulated actions may not prevent your test device from auto-locking. To ensure that auto-locking does not happen, before running tests on a device, you should set the Auto-Lock setting to Never.

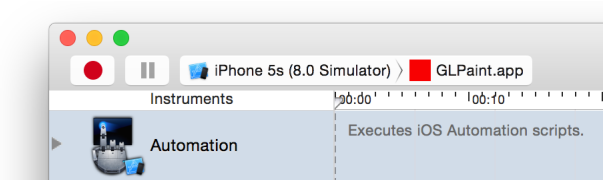
iOS 8 Enhancement: iOS 8 includes a new Enable UI Automation preference under Settings > Developer, which allows third-party developers finer control of when their devices are available to perform automation. For physical iOS devices, this setting is off by default and must be enabled prior to performing any UI Automation. In the simulator, the setting is enabled by default.

Writing, Exporting, and Importing Automation Test Scripts

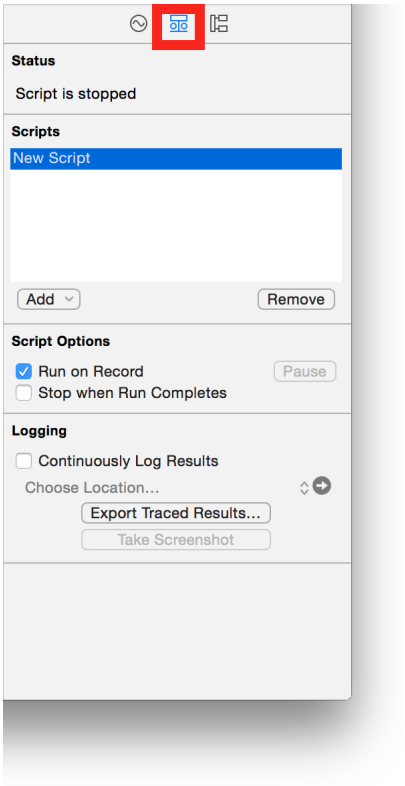
It's easy to write your own scripts inside Instruments. The built-in script editor in the Automation instrument allows you to create and edit new test scripts in your trace document, as well as import existing ones.

To create a new script

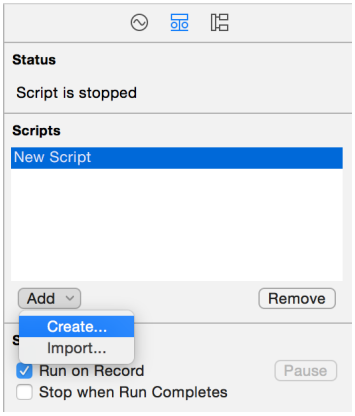
1. Create a new trace document in Instruments using the Automation trace template.



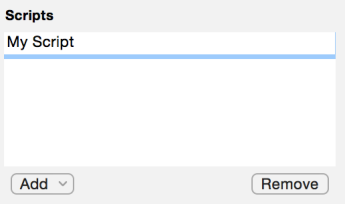
2. With the Automation instrument selected in the Instruments pane, click the Display Settings button (middle) in the inspector sidebar.



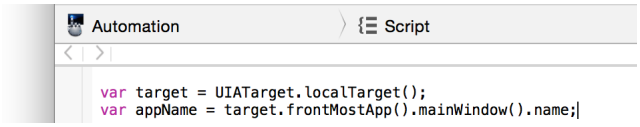
3. Click Add > Create.



4. Double-click New Script to change the name of the script.



5. In the Detail pane Navigation bar, select Script to enter the code for your script.



6. Choose a target for your script.

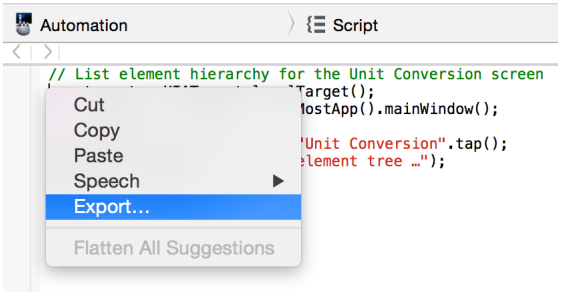
7. Click the Play button at the bottom of the the Automation > Script Detail pane.

After you create a script, you will want to use it throughout the development of your app. You do this by saving your configured trace document (which includes your script) and opening it again whenever you want to test your app. Or, you can export your test script and import it into a new trace document when you need it.

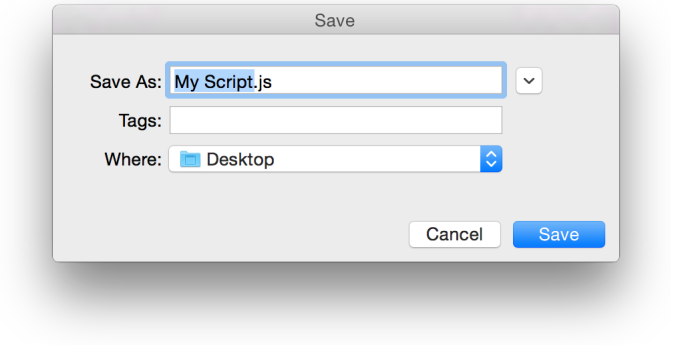
To export a script to a file on a disk

1. Create a script in a trace document.
2. Control-click in the content area to display the contextual menu.

3. Choose Export.

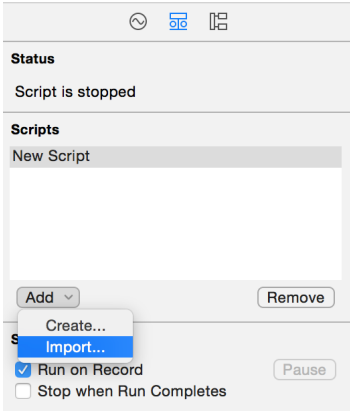


4. Choose a location for your script in the file system and click Save.



To import a previously saved script

1. Select the Automation trace template.
2. Click Add > Import in the Scripts area of the Display Settings inspector.



3. Navigate to your saved script file and click Open.

Loading Saved Automation Test Scripts

You write your Automation tests in JavaScript, using the UI Automation JavaScript library to specify actions that should be performed in your app as it runs. You can create as many scripts as you like and include them in your trace document, but you can run only one script at a time. The API does, however, offer a `#import` directive that allows you to write smaller, reusable discrete test scripts. For example, if you define commonly used functions in a file named `TestUtilities.js`, you can make those functions available for use in your test script by including in that script the line:

```
#import "<path-to-library-folder>/TestUtilities.js"
```

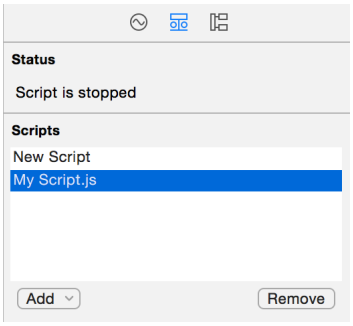
Changes you make with the script editor are saved when you save your trace document. For scripts created in the editor, changes are saved as part of the trace document itself. To save those changes in a file you can access on disk, you have to export the script. See To export a script to a file on a disk (page 96) above.

Recording Manual User Interface Actions into Automation Scripts

A capture feature simplifies script development by allowing you to record actions that you perform on a target iOS device or in iOS Simulator. To use this feature, create an Automation trace document and then capture actions that you perform on the device. These captured actions are incorporated into your script as expressions that you can edit.

To record manual user interface actions

1. Create or open a trace document containing the Automation instrument.
2. Click the Display Settings button in the inspector sidebar, if necessary, to display the Scripts display settings.
3. Select your script from the list.



> **Note:** If your script is not in the list, you can import it (choose Add > Import) or create a new one (choose Add > Create).

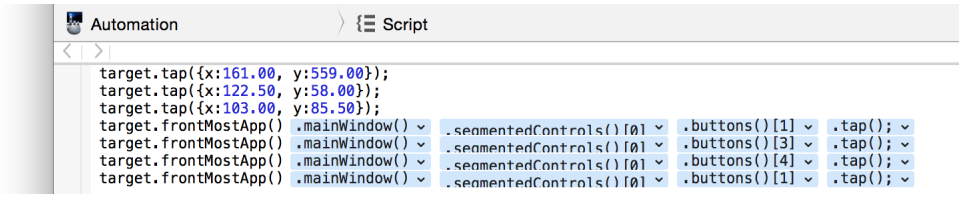
4. Click in the script editor pane to position the cursor where you want the captured actions to appear in the script.
5. Click the Record button under the text editor.

The target application launches, and the script status is updated to indicate that capturing is in progress.

6. Perform the desired actions on the device or in the simulator.

> **Important:** To ensure accurate capture, perform these actions slowly and precisely.

7. Click the Stop button under the text editor to stop capturing actions.



The Automation instrument generates expressions in your script for the actions you perform. Some of these expressions include tokens that contain alternative syntax for the expression. To see the alternative syntax, select the arrow at the right of the token. To select the currently displayed syntax for a token and flatten the expression, double-click the token.

To configure the Automation instrument to automatically start and stop your script under control of the Instruments Record button in the toolbar, select the “Run on Record” checkbox.

If your app crashes or goes to the background, your script is blocked until the app is frontmost again, at which time the script continues to run.

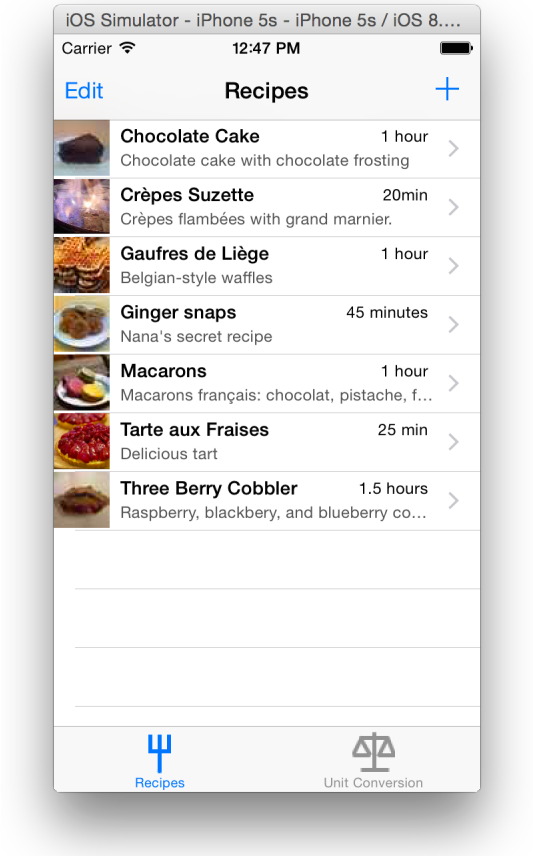
Important: You must explicitly stop recording, either with the Stop button or by selecting Stop when Run Completes (not the default). Completion or termination of your script does not turn off recording.

Accessing and Manipulating UI Elements

The Accessibility-based mechanism underlying the UI Automation feature represents every control in your app as a uniquely identifiable element. To perform an action on an element in your app, you explicitly identify that element in terms of the app's element hierarchy. To fully understand this section, you should be familiar with the information in *iOS Human Interface Guidelines* .

To illustrate the element hierarchy, this section refers to the Recipes iOS app shown in Figure 11-1, which is available as the code sample *iPhoneCoreDataRecipes* from the iOS Dev Center.

Figure 11-1 The Recipes app (Recipes screen)

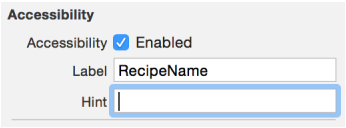


UI Element Accessibility

Each accessible element is inherited from the base element, `UIAElement`. Every element can contain zero or more other elements.

As detailed below, your script can access individual elements by their position within the element hierarchy. However, you can assign a unique name to each element by setting the label attribute and making sure Accessibility is selected in Interface Builder for the control represented by that element, as shown in Figure 11-2.

Figure 11-2 Setting the accessibility label in Interface Builder



UI Automation uses the accessibility label (if it's set) to derive a name property for each element. Aside from the obvious benefits, using such names can greatly simplify development and maintenance of your test scripts.

The name property is one of four properties of these elements that can be very useful in your test scripts.

- **name.** Derived from the accessibility label
- **value.** The current value of the control, for example, the text in a text field
- **elements.** Any child elements contained within the current element, for example, the cells in a table view
- **parent.** The element that contains the current element

Understanding the Element Hierarchy

At the top of the element hierarchy is the `UIATarget` class, which represents the high-level user interface elements of the system under test (SUT)—that is, the device (or simulator) as well as the iOS and your app running on that device. For the purposes of your test, your app is the frontmost app (or target app), identified as follows:

```
UIATarget.localTarget().frontMostApp();
```

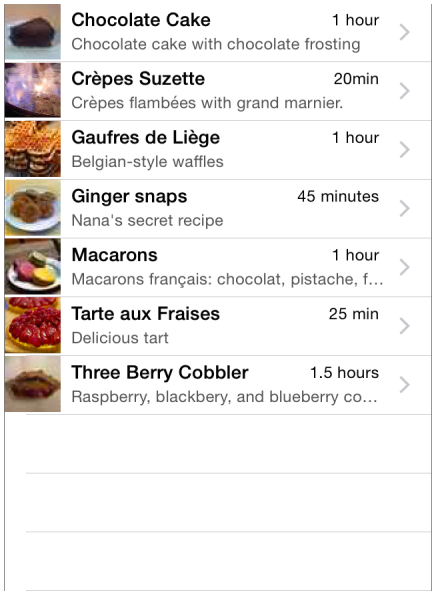
To reach the app window, the main window of your app, you would specify

```
UIATarget.localTarget().frontMostApp().mainWindow();
```

At startup, the Recipes app window appears as shown in Figure 11-1 (page 100).

Inside the window, the recipe list is presented in an individual view, in this case, a table view, see Figure 11-3.

Figure 11-3 Recipes table view



This is the first table view in the app's array of table views, so you specify it as such using the zero index ([0]), as follows:

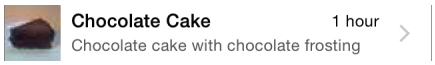
```
UIATarget.localTarget().frontMostApp().mainWindow().tableViews()[0];
```

Inside the table view, each recipe is represented by a distinct individual cell. You can specify individual cells in similar fashion. For example, using the zero index ([0]), you can specify the first cell as follows:

```
UIATarget.localTarget().frontMostApp().mainWindow().tableViews()[0].cells()[0];
```

Each of these individual cell elements is designed to contain a recipe record as a custom child element. in this first cell is the record for chocolate cake, which you can access by name with this line of code:

```
UIATarget.localTarget().frontMostApp().mainWindow().tableViews()[0].cells()[0].elements()["Chocolate
 Cake"];
```



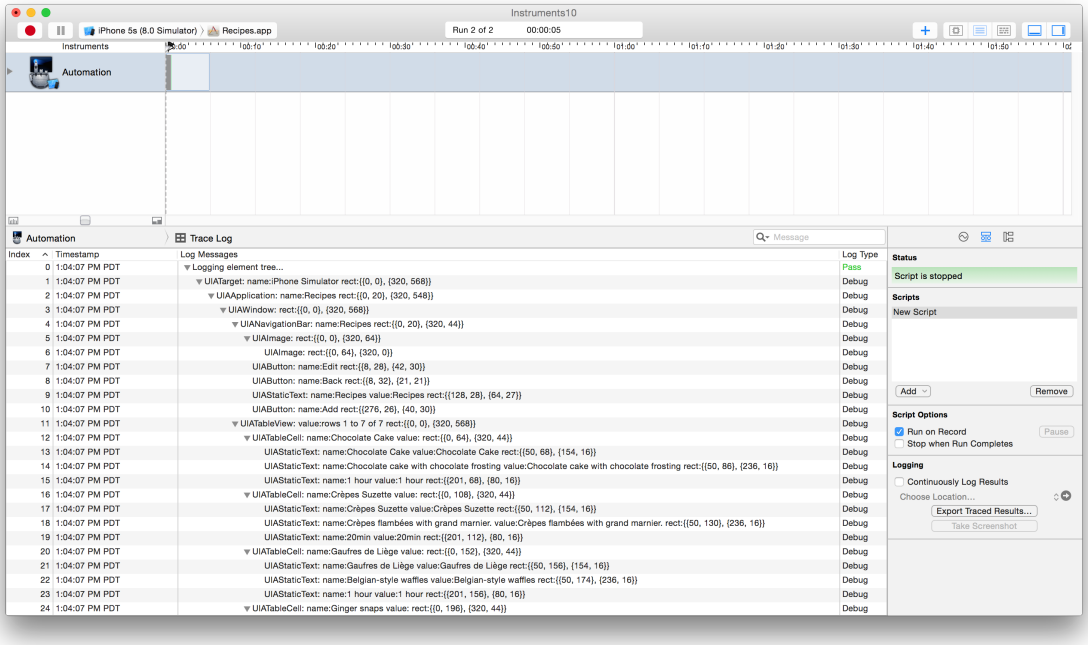
Displaying the Element Hierarchy

You can use the `logElementTree` method for any element to list all of its child elements. The following code illustrates listing the elements for the main (Recipes) screen (or mode) of the Recipes app.

```
// List element hierarchy for the Recipes screen
UIALogger.logStart("Logging element tree ...");
UIATarget.localTarget().logElementTree();
UIALogger.logPass();
```

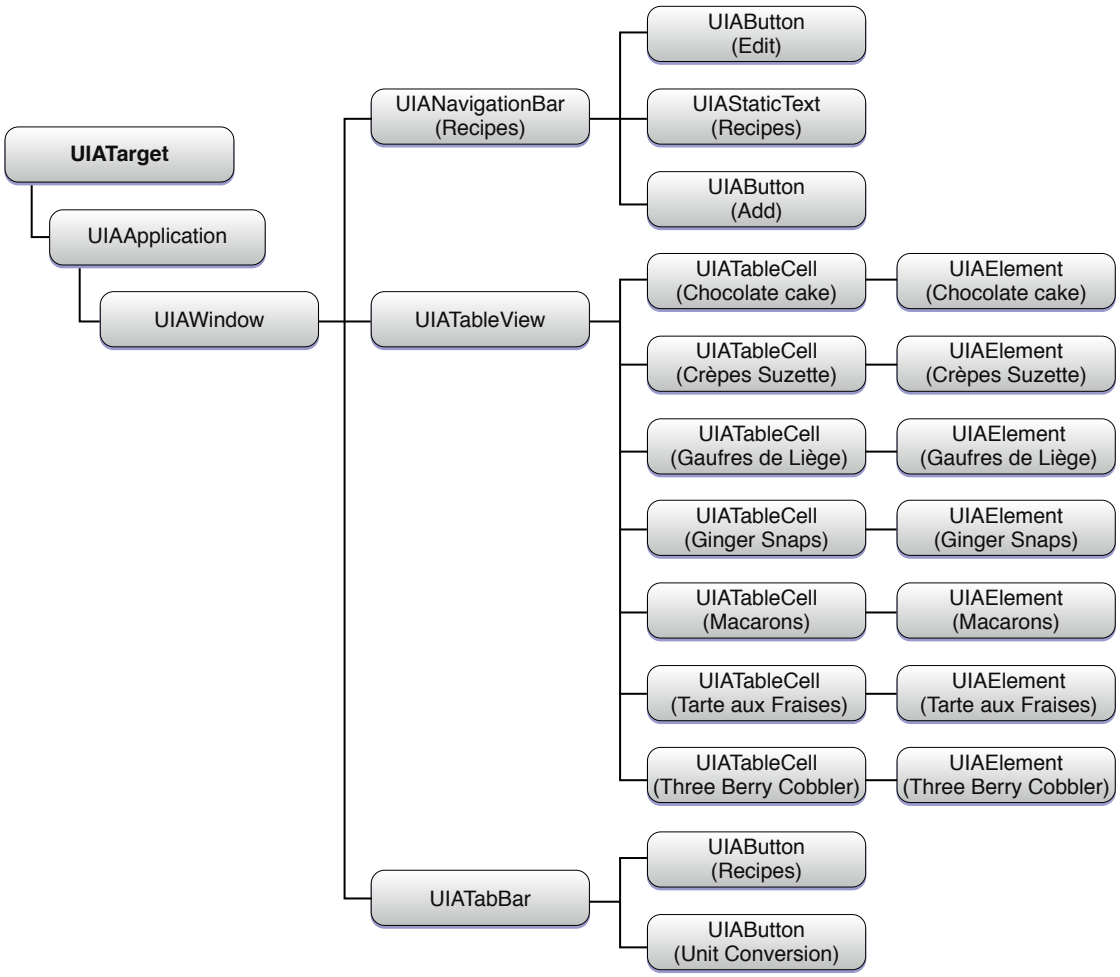
The output of the command is captured in the log displayed by the Automation instrument, as in Figure 11-4.

Figure 11-4 Output from the `logElementTree` method



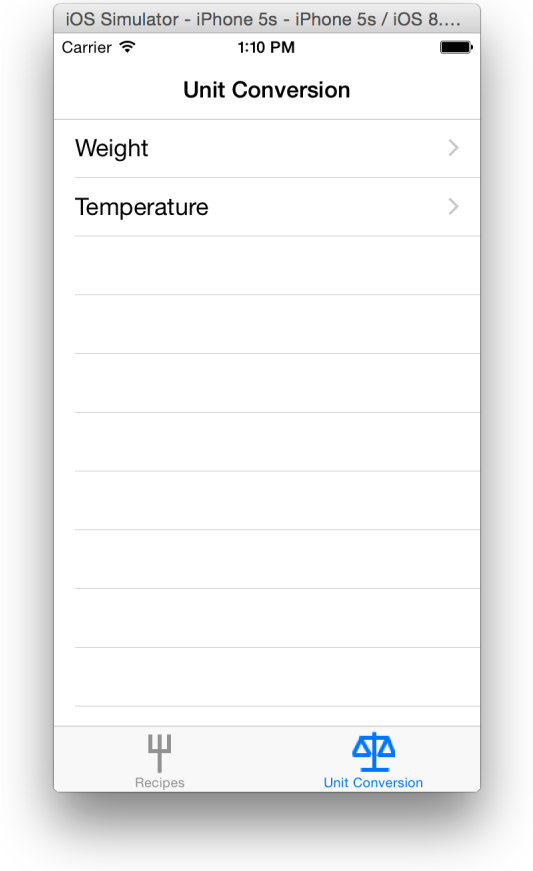
Note the indentation of each element line item, indicating that element's level in the hierarchy. These levels may be viewed conceptually, as in Figure 11-5.

Figure 11-5 Element hierarchy (Recipes screen)



Although a screen is not technically an iOS programmatic construct and doesn't explicitly appear in the hierarchy, it is a helpful concept in understanding that hierarchy. Tapping the Unit Conversion tab in the tab bar displays the Unit Conversion screen (or mode), shown in Figure 11-6.

Figure 11-6 Recipes app (Unit Conversion screen)

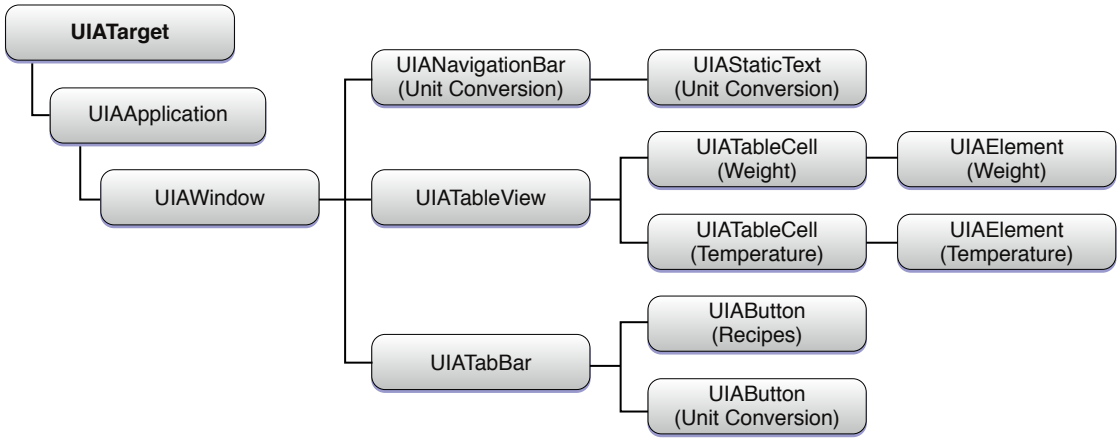


The following code taps the Unit Conversion tab in the tab bar to display the associated screen and then logs the element hierarchy associated with it:

```
// List element hierarchy for the Unit Conversion screen
var target = UIATarget.localTarget();
var appWindow = target.frontMostApp().mainWindow();
var element = target;
appWindow.tabBar().buttons()["Unit Conversion"].tap();
UIALogger.logStart("Logging element tree …");
element.logElementTree();
UIALogger.logPass();
```

The resulting log reveals the hierarchy to be as illustrated in Figure 11-7. Just as with the previous example, `logElementTree` is called for the target, but the results are for the current screen—in this case, the Unit Conversion screen.

Figure 11-7 Element hierarchy (Unit Conversion screen)



Simplifying Element Hierarchy Navigation

The previous code sample introduces the use of variables to represent parts of the element hierarchy. This technique allows for shorter, simpler commands in your scripts.

Using variables in this way also allows for some abstraction, yielding flexibility in code use and reuse. The following example uses a variable (`destinationScreen`) to control changing between the two main screens (Recipes and Unit Conversion) of the Recipes app:

```
// Switch screen (mode) based on value of variable
var target = UIATarget.localTarget();
var app = target.frontMostApp();
var tabBar = app.mainWindow().tabBar();
var destinationScreen = "Recipes";
if (tabBar.selectedButton().name() != destinationScreen) {
    tabBar.buttons()[destinationScreen].tap();
}
```

With minor variations, this code could work, for example, for a tab bar with more tabs or with tabs of different names.

Performing User Interface Gestures

Once you understand how to access the desired element, it's relatively simple and straightforward to manipulate that element.

The UI Automation API provides methods to perform most UIKit user actions, including multi-touch gestures. For comprehensive detailed information about these methods, see *UI Automation JavaScript Reference for iOS*.

Tapping. Perhaps the most common touch gesture is a simple tap. Implementing a one-finger single tap on a known UI element is very simple. For example, tapping the right button, labeled with a plus sign (+), in the navigation bar of the Recipes app, displays a new screen used to add a new recipe.

This command is all that's required to tap that button:

```
UIATarget.localTarget().frontMostApp().navigationBar().buttons()["Add"].tap();
```

Note that it uses the name *Add* to identify the button, presuming that the accessibility label has been set appropriately, as described above.

Of course, more complicated tap gestures are required to thoroughly test any sophisticated app. You can specify any standard tap gestures. For example, to tap once at an arbitrary location on the screen, you just need to provide the screen coordinates:

```
UIATarget.localTarget().tap({x:100, y:200});
```

This command taps at the x and y coordinates specified, regardless of what's at that location on the screen.

More complex taps are also available. To double-tap the same location, you could use this code:

```
UIATarget.localTarget().doubleTap({x:100, y:200});
```

And to perform a two-finger tap to test zooming in and out, for example, you could use this code:

```
UIATarget.localTarget().twoFingerTap({x:100, y:200});
```

Pinching. A pinch open gesture is typically used to zoom in or expand an object on the screen, and a pinch close gesture is used for the opposite effect—to zoom out or shrink an object on the screen. You specify the coordinates to define the start of the pinch close gesture or end of the pinch open gesture, followed by a number of seconds for the duration of the gesture. The duration parameter allows you some flexibility in specifying the speed of the pinch action.

```
UIATarget.localTarget().pinchOpenFromToForDuration({x:20, y:200}, {x:300, y:200},
 2);
```

```
UIATarget.localTarget().pinchCloseFromToForDuration({x:20, y:200}, {x:300, y:200},
 2);
```

Dragging and flicking. If you need to scroll through a table or move an element on screen, you can use the `dragFromToForDuration` method. You provide coordinates for the starting location and ending location, as well as a duration, in seconds. The following example specifies a drag gesture from location 160, 200 to location 160, 400, over a period of 1 second:

```
UIATarget.localTarget().dragFromToForDuration({x:160, y:200}, {x:160, y:400}, 1);
```

A flick gesture is similar, but it is presumed to be a fast action, so it doesn't require a duration parameter.

```
UIATarget.localTarget().flickFromTo({x:160, y:200}, {x:160, y:400});
```

Entering text. Your script will likely need to test that your app handles text input correctly. To do so, it can enter text into a text field by simply specifying the target text field and setting its value with the `setValue` method. The following example uses a local variable to provide a long string as a test case for the first text field (index [0]) in the current screen:

```
var recipeName = "Unusually Long Name for a Recipe";
UIATarget.localTarget().frontMostApp().mainWindow().textFields()[0].setValue(recipeName);
```

Navigating in your app with tabs. To test navigating between screens in your app, you'll very likely need to tap a tab in a tab bar. Tapping a tab is much like tapping a button; you access the appropriate tab bar, specify the desired button, and tap that button, as shown in the following example:

```
var tabBar = UIATarget.localTarget().frontMostApp().mainWindow().tabBar();
var selectedTabName = tabBar.selectedButton().name();
```

```
if (selectedTabName != "Unit Conversion")  {
    tabBar.buttons()["Unit Conversion"].tap();
}
```

First, a local variable is declared to represent the tab bar. Using that variable, the script accesses the tab bar to determine the selected tab and get the name of that tab. Finally, if the name of the selected tab matches the name of the desired tab (in this case “Unit Conversion”), the script taps that tab.

Scrolling to an element. Scrolling is a large part of a user’s interaction with many apps. UI Automation provides a variety of methods for scrolling. The basic methods allow for scrolling to the next element left, right, up, or down. More sophisticated methods support greater flexibility and specificity in scrolling actions. One such method is `scrollToElementWithPredicate`, which allows you to scroll to an element that meets certain criteria that you specify. This example accesses the appropriate table view through the element hierarchy and scrolls to a recipe in that table view whose name starts with “Turtle Pie.”

```
UIATarget.localTarget().frontMostApp().mainWindow().tableViews()[0] \
    .scrollToElementWithPredicate("name beginswith 'Turtle Pie'");
```

Using the `scrollToElementWithPredicate` method allows scrolling to an element whose exact name may not be known.

Using predicate functionality can significantly expand the capability and applicability of your scripts. For more information on using predicates, see *Predicate Programming Guide*.

Other useful methods for flexibility in scrolling include `scrollToElementWithName` and `scrollToElementWithValueForKey`. See *UIAScrollView Class Reference* for more information.

Accessibility Label and Identifier Attributes

The label attribute and identifier attribute figure prominently in your script’s ability to access UI elements, so it’s a good idea to understand how they are used.

Setting a meaningful value for the label attribute is optional but recommended. You can set and view the label string in the Label text field in the Accessibility section of the Identity inspector in Interface Builder. This label is expected to be descriptive but short, partly because assistive technologies such as Apple’s VoiceOver use it as the name of the associated UI element. In UI Automation, this label is returned by the `label` method. It is also returned by the `name` method as a default if the `identifier` attribute is not set. For an overview of accessibility labels, see *Tic Tac Toe: Creating Accessible Apps with Custom UI* and *Accessibility Programming Guide for iOS*. For reference details, see *UIAccessibilityElement Class Reference*.

The identifier attribute allows you to use more descriptive names for elements. It is optional, but it must be set for the script to perform either of these two operations:

- Accessing a container view by name while also being able to access its children
- Accessing a UILabel view by name to obtain its displayed text (through its `value` attribute)

In UI Automation, the `name` method returns the value of this identifier attribute, if one is set. If it is not set, the `name` method returns the value of the `label` attribute.

Currently, you can set a value for the identifier attribute only programmatically, through the `accessibilityIdentifier` property. For details, see *UIAccessibilityIdentification Protocol Reference*.

Adding Timing Flexibility with Timeout Periods

While executing a test script, an attempt to access an element can fail for a variety of reasons. For example, an action could fail if:

- The app is still in the process of launching.
- A new screen hasn't yet been completely drawn.
- The element (such as a button your script is trying to click) may be drawn, but its contents are not filled in or updated yet.

In situations like these, your script may need to wait for some action to complete before proceeding. In the Recipes app, for example, the user taps the Recipes tab to return from the Unit Conversion screen to the Recipes screen. However, UI Automation may detect the existence of the Add button, enabling the test script to attempt to tap it—before the button is actually drawn and the app is actually ready to accept that tap. An accurate test must ensure that the Recipes screen is completely drawn and that the app is ready to accept user interaction with the controls within that screen before proceeding.

To provide some flexibility in such cases and to give you finer control over timing, UI Automation provides for a timeout period, a period during which it repeatedly attempts to perform the specified action before failing. If the action completes during the timeout period, that line of code returns, and your script can proceed. If the action doesn't complete during the timeout period, an exception is thrown and UI Automation returns a `UIAElementNil` object. A `UIAElementNil` object is always considered invalid.

The default timeout period is five seconds, but your script can change that at any time. For example, you might decrease the timeout period if you want to test whether an element exists but don't need to wait if it isn't. On the other hand, you might increase the timeout period when the script must access an element but the user interface is slow to update. The following methods for manipulating the timeout period are available in the UIATarget class:

- `timeout`: Returns the current timeout value.
- `setTimeout`: Sets a new timeout value.
- `pushTimeout`: Stores the current timeout value on a stack and sets a new timeout value.
- `popTimeout`: Retrieves the previous timeout value from a stack, restores it as the current timeout value, and returns it.

To make this feature as easy as possible to use, UI Automation uses a stack model. You push a custom timeout period to the top of the stack, as with the following code that shortens the timeout period to two seconds.

```
UIATarget.localTarget().pushTimeout(2);
```

You then run the code to perform the action and pop the custom timeout off the stack.

```
UIATarget.localTarget().popTimeout();
```

Using this approach you end up with a robust script, waiting a reasonable amount of time for something to happen.

> **Note:** Although using explicit delays is typically not encouraged, on occasion it may be necessary. The following code shows how you specify a delay of 2 seconds:
>
> ```
> UIATarget.localTarget().delay(2);
> ```

For more details, see *UIATarget Class Reference*.

> **Note:** A timeout value of `0` immediately returns a `UIAElementNil` object if the initial attempt to access the element fails.

Logging Test Results and Data

Your script reports log information to the Automation instrument, which gathers it and reports it back for analysis.

When writing your tests, you should log as much information as you can, if just to help you diagnose any failures that occur. At a bare minimum, you should log when each test begins and ends, identifying the test performed and recording pass/fail status. This kind of minimal logging is almost automatic in UI Automation. You simply call `logStart` with the name of your test, run your test, then call `logPass` or `logFail` as appropriate, as shown in the following example:

```
var testName = "Module 001 Test";
UIALogger.logStart(testName);
//some test code
UIALogger.logPass(testName);
```

But it's a good practice to log what transpires whenever your script interacts with a control. Whether you're validating that parts of your app perform properly or you're still tracking down bugs, it's hard to imagine having too much log information to analyze. To this end, you can log just about any occurrence using `logMessage`, and you can even supplement the textual data with screenshots.

The following code example expands the logging of the previous example to include a free-form log message and a screenshot:

```
var testName = "Module 001 Test";
UIALogger.logStart(testName);
//some test code
UIALogger.logMessage("Starting Module 001 branch 2, validating input.");
//capture a screenshot with a specified name
UIATarget.localTarget().captureScreenWithName("SS001-2_AddedIngredient");
//more test code
UIALogger.logPass(testName);
```

The screenshot requested in the example would be saved back to Instruments and appear in the Editor Log in the detail pane with the specified filename (`SS001-2_AddedIngredient.png`, in this case).

Using Screenshots

Your script can capture screenshots using the `captureScreenWithName` and `captureRectWithName` methods in the `UIATarget` class. To ensure easy access to those screenshots, open the Logging section at the left of the template, select the Continuously Log Results option, and use the Choose Location pop-up menu to specify a folder for the log results. Each captured screenshot is stored in the results folder with the name specified by your script.

> **Note:** The results folder pathname can be set from the command line using the `UIARESULTSPATH` environment variable. Setting this variable overrides the results folder setting in the trace template.

Verifying Test Results

The crux of testing is being able to verify that each test has been performed and that it has either passed or failed. This code example runs the test `testName` to determine whether a valid element recipe element whose name starts with "Tarte" exists in the recipe table view. First, a local variable is used to specify the cell criteria:

```
var cell = UIATarget.localTarget().frontMostApp().mainWindow() \
    .tableViews()[0].cells().firstWithPredicate("name beginswith 'Tarte'");
```

Next, the script uses the `isValid` method to test whether a valid element matching those criteria exists in the recipe table view.

```
if (cell.isValid()) {
    UIALogger.logPass(testName);
}
else {
    UIALogger.logFail(testName);
}
```

If a valid cell is found, the code logs a pass message for the `testName` test; if not, it logs a failure message.

Notice that this test specifies `firstWithPredicate` and `"name beginsWith 'Tarte'"`. These criteria yield a reference to the cell for “Tarte aux Fraises,” which works for the default data already in the Recipes sample app. If, however, a user adds a recipe for “Tarte aux Framboises,” this example may or may not give the desired results.

Handling Alerts

In addition to verifying that your app’s alerts perform properly, your test should accommodate alerts that appear unexpectedly from outside your app. For example, it’s not unusual to get a text message while checking the weather or playing a game.

Handling Externally Generated Alerts

Although it may seem somewhat paradoxical, your app and your tests should expect that unexpected alerts will occur whenever your app is running. Fortunately, UI Automation includes a default alert handler that renders external alerts very easy for your script to cope with. Your script provides an alert handler function called `onAlert`, which is called when the alert has occurred, at which time it can take any appropriate action, and then then simply return the alert to the default handler for dismissal.

The following code example illustrates a very simple alert case:

```
UIATarget.onAlert = function onAlert(alert) {
    var title = alert.name();
    UIALogger.logWarning("Alert with title '" + title + "' encountered.");
    // return false to use the default handler
    return false;
}
```

All this handler does is log a message that this type of alert was received and then return `false`. Returning `false` directs the UI Automation default alert handler to just dismiss the alert. In the case of an alert for a received text message, for example, UI Automation clicks the Close button.

> **Note:** The default handler stops dismissing alerts after reaching an upper limit of sequential alerts. In the unlikely case that your test reaches this limit, you should investigate possible problems with your testing environment and procedures.

Handling Internally Generated Alerts

As part of your app, you will have alerts that need to be handled. In those instances, your alert handler needs to perform the appropriate response and return `true` to the default handler, indicating that the alert has been handled.

The following code example expands slightly on the basic alert handler. After logging the alert type, it tests whether the alert is the specific one that's anticipated. If so, it taps the Continue button, which is known to exist, and returns `true` to skip the default dismissal action.

```
UIATarget.onAlert = function onAlert(alert) {
    var title = alert.name();
    UIALogger.logWarning("Alert with title '" + title + "' encountered.");
    if (title == "The Alert We Expected") {
        alert.buttons()["Continue"].tap();
        return true;  //alert handled, so bypass the default handler
    }
    // return false to use the default handler
    return false;
}
```

This basic alert handler can be generalized to respond to just about any alert received, while allowing your script to continue running.

Detecting and Specifying Device Orientation

A well-behaved iOS app is expected to handle changes in device orientation gracefully, so your script should anticipate and test for such changes.

UI Automation provides `setDeviceOrientation` to simulate a change in the device orientation. This method uses the constants listed in Table 11-1.

Note: With regard to device orientation handling, it bears repeating that the functionality is entirely simulated in software. Hardware features such as raw accelerometer data are both unavailable to this UI Automation feature and unaffected by it.

Table 11-1 Device orientation constants

Orientation constant	Description
UIA_DEVICE_ORIENTATION_UNKNOWN	The orientation of the device cannot be determined.
UIA_DEVICE_ORIENTATION_PORTRAIT	The device is in portrait mode, with the device upright and the home button at the bottom.
UIA_DEVICE_ORIENTATION_PORTRAIT_UPSIDEDOWN	The device is in portrait mode but upside down, with the device upright and the home button at the top.
UIA_DEVICE_ORIENTATION_LANDSCAPELEFT	The device is in landscape mode, with the device upright and the home button on the right side.
UIA_DEVICE_ORIENTATION_LANDSCAPERIGHT	The device is in landscape mode, with the device upright and the home button on the left side.
UIA_DEVICE_ORIENTATION_FACEUP	The device is parallel to the ground with the screen facing upward.
UIA_DEVICE_ORIENTATION_FACEDOWN	The device is parallel to the ground with the screen facing downward.

In contrast to device orientation is interface orientation, which represents the rotation required to keep your app's interface oriented properly upon device rotation. Note that in landscape mode, device orientation and interface orientation are opposite because rotating the device requires rotating the content in the opposite direction.

UI Automation provides the `interfaceOrientation` method to get the current interface orientation. This method uses the constants listed in Table 11-2.

Table 11-2 Interface orientation constants

Orientation constant	Description
UIA_INTERFACE_ORIENTATION_PORTRAIT	The interface is in portrait mode, with the bottom closest to the home button.
UIA_INTERFACE_ORIENTATION_PORTRAIT_UPSIDEDOWN	The interface is in portrait mode but upside down, with the top closest to the home button.
UIA_INTERFACE_ORIENTATION_LANDSCAPELEFT	The interface is in landscape mode, with the left side closest to the home button.
UIA_INTERFACE_ORIENTATION_LANDSCAPERIGHT	The interface is in landscape mode, with the right side closest to the home button.

The following example changes the device orientation (in this case, to landscape left), then changes it back (to portrait):

```
var target = UIATarget.localTarget();
var app = target.frontMostApp();
//set orientation to landscape left
target.setDeviceOrientation(UIA_DEVICE_ORIENTATION_LANDSCAPELEFT);
UIALogger.logMessage("Current orientation now " + app.interfaceOrientation());
//reset orientation to portrait
target.setDeviceOrientation(UIA_DEVICE_ORIENTATION_PORTRAIT);
UIALogger.logMessage("Current orientation now " + app.interfaceOrientation());
```

Of course, once you've rotated, you do need to rotate back again.

When performing a test that involves changing the orientation of the device, it is a good practice to set the rotation at the beginning of the test, then set it back to the original rotation at the end of your test. This practice ensures that your script is always back in a known state.

You may have noticed the orientation logging in the example. Such logging provides additional assurance that your tests—and your testers—don't become disoriented.

Testing for Multitasking

When a user exits your app by tapping the Home button or causing some other app to come to the foreground, your app is suspended. To simulate this occurrence, UI Automation provides the `deactivateAppForDuration` method. You just call this method, specifying a duration, in seconds, for which your app is to be suspended, as illustrated by the following example:

```
UIATarget.localTarget().deactivateAppForDuration(10);
```

This single line of code causes the app to be deactivated for 10 seconds, just as though a user had exited the app and returned to it 10 seconds later.

Running a Test Script from an Xcode Project

You can easily automate running your test script by creating a custom Automation instrument template.

Creating a Custom Automation Instrument Template

To create a custom Automation instrument template:

1. Launch the Instruments app.
2. Choose the Automation template to create a trace document.

   > **Note:** Alternatively, you can add the Automation instrument to an existing trace template from the UI Automation group of the instrument library and drag it to your trace document.

3. Choose View > Detail, if necessary, to display the detail view.
4. Select your script from the list.

   > **Note:** If your script is not in the list, you can import it (choose Add > Import) or create a new one (choose Add > Create).

5. Edit your script as needed in the Script area of the detail pane.
6. Choose File > Save as Template, name the template, and save it to the default Instruments template location:

   `~/Library/Application Support/Instruments/Templates/`

Executing an Automation Instrument Script in Xcode

After you have created your customized Automation template, you can execute your test script from Xcode by following these steps:

1. Open your project in Xcode.
2. From the Scheme pop-up menu (in the workspace window toolbar), select Edit Scheme for a scheme with which you would like to use your script.
3. Select Profile from the left column of the scheme editing dialog.
4. Choose your application from the Executable pop-up menu.
5. Choose your customized Automation Instrument template from the Instrument pop-up menu.
6. Click OK to approve your changes and dismiss the scheme editor dialog.
7. Choose Product > Profile.

   Instruments launches and executes your test script.

Executing an Automation Instrument Script from the Command Line

You can also execute your test script from the command line. If you have created a customized Automation template as described in Creating a Custom Automation Instrument Template (page 118), you can use the following simple command:

```
instruments –w deviceID –t templateFilePath targetAppName
```

`deviceID`
: The 40-character device identifier, available in the Xcode Devices organizer, and in iTunes.

Note: Omit the device identifier option (`–w deviceID` in this example) to target the Simulator instead of a device.

`templateFilePath`
: The full pathname of your customized Automation template, by default, `~/Library/Application Support/Instruments/Templates/templateName`, where `templateName` is the name you saved it with.

`targetAppName`
: The local name of the application. When targeting a device, omit the pathname and `.app` extension. When targeting a simulator, use the full pathname.

You can use the default trace template if you don't want to create a custom one. To do so, you use the environment variables `UIASCRIPT` and `UIARESULTSPATH` to identify the script and the results directory.

```
instruments –w deviceID –t defaultTemplateFilePath targetAppName \
   –e UIASCRIPT scriptFilePath –e UIARESULTSPATH resultsFolderPath
```

`defaultTemplateFilePath`

The full pathname of the default template:

```
/Applications/Xcode.app/Contents/Developer/Platforms/iPhoneOS.platform/Developer/Library/Instruments/PlugIns/
AutomationInstrument.bundle/Contents/Resources/Automation.tracetemplate
```

`scriptFilePath`

The file-system location of your test script.

`resultsFolderPath`

The file-system location of the directory to hold the results of your test script.

Creating Custom Instruments

You've learned that Instruments has built-in instruments that provide a great deal of information about the inner workings of your app. Sometimes, though, you may want to tailor the information being gathered more closely to your own code. For example, instead of gathering data every time a function is called, you might want to set conditions on when data is gathered. Alternatively, you might want to dig deeper into your own code than the built-in instruments allow. For these situations, Instruments lets you create custom instruments. Whenever possible, it is recommended that you use an existing instrument instead of creating a new instrument. Creating custom instruments is an advanced feature.

The following sections show you how to create a custom instrument and how to use that instrument both with the Instruments app and with the `dtrace` command-line tool.

About Custom Instruments

Custom instruments use DTrace for their implementation. DTrace is a dynamic tracing facility originally created by Sun and ported to OS X. Because DTrace taps into the operating system kernel, you have access to low-level information about the kernel itself and the user processes running on your computer. Many of the built-in instruments are already based on DTrace. And even though DTrace is itself a very powerful and complex tool, Instruments provides a simple interface that gives you access to the power of DTrace without the complexity.

DTrace has not been ported to iOS, so it is not possible to create custom instruments for devices running iOS.

Important: Although the custom instrument builder simplifies the process of creating DTrace probes, you should still be familiar with DTrace and how it works before creating new instruments. Many of the more powerful debugging and data gathering actions require you to write DTrace scripts. To learn about DTrace and the D scripting language, see the *Solaris Dynamic Tracing Guide* , available from the Oracle Technology Network. For information about the `dtrace` command-line tool, see the `dtrace` man page.

Note: Several Apple apps—namely, iTunes, DVD Player, and apps that use QuickTime—prevent the collection of data through DTrace (either temporarily or permanently) in order to protect sensitive and copyrighted data. Therefore, you should not run those apps when performing systemwide data collection.

Custom instruments are built using DTrace probes. A probe is like a sensor that you place in your code. It corresponds to a location or event, such as a function entry point, to which DTrace can bind. When the function executes or the event is generated, the associated probe fires and DTrace runs whatever actions are associated with the probe. Most DTrace actions simply collect data about the operating system and user app behavior at that moment. It is possible, however, to run custom scripts as part of an action. Scripts let you use the features of DTrace to fine tune the data you gather.

Probes fire each time they are encountered, but the action associated with the probe need not be run every time the probe fires. A predicate is a conditional statement that allows you to restrict when the probe's action is run. For example, you can restrict a probe to a specific process or user, or you can run an action when a specific condition in your instrument is true. By default, probes do not have any predicates, meaning that the associated action runs every time the probe fires. You can add any number of predicates to a probe, however, and link them together using `AND` and `OR` operators to create complex decision trees.

A custom instrument consists of the following blocks:

- A description block, containing the name, category, and description of the instrument
- One or more probes, each containing its associated actions and predicates
- A DATA declaration area, for declaring global variables shared by all probes
- A BEGIN script, which initializes any global variables and performs any startup tasks required by the instrument
- An END script, which performs any final cleanup actions

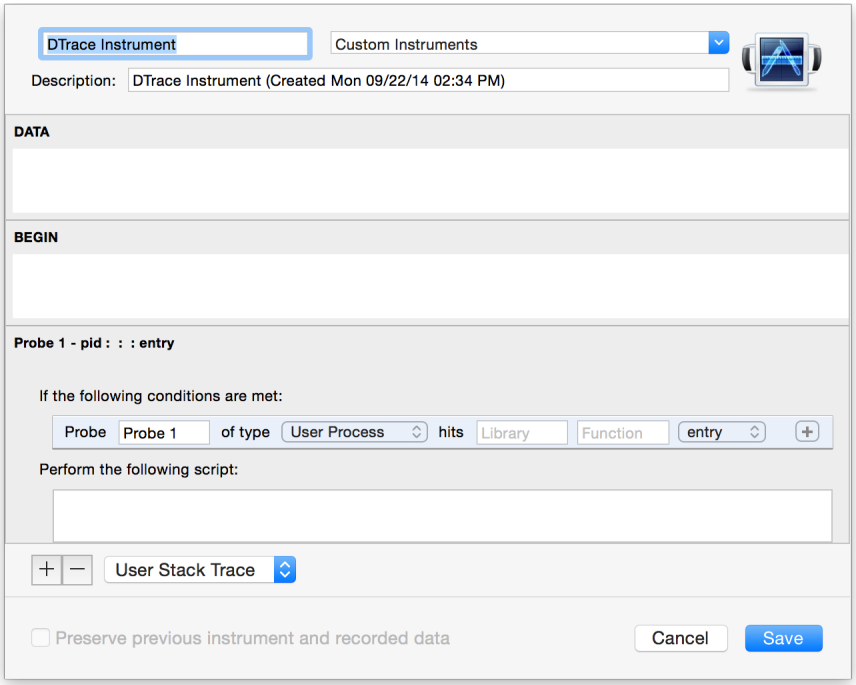
All instruments must have at least one probe with its associated actions. Similarly, all instruments should have an appropriate name and description to identify them to Instruments users. Instruments displays your instrument's descriptive information in the library window. Providing good information makes it easier to remember what the instrument does and how it should be used.

Probes are not required to have global DATA or BEGIN and END scripts. Those elements are used in advanced instrument design when you want to share data among probes or provide some sort of initial configuration for your instrument. The creation of DATA, BEGIN, and END blocks is described in Tips for Writing Custom Scripts (page 131).

Creating a Custom Instrument

To create a custom DTrace instrument, select Instrument > Build New Instrument. This command displays the instrument configuration sheet, shown in Figure 12-1. You use this sheet to specify your instrument information, including any probes and custom scripts.

Figure 12-1 The instrument configuration sheet



At a minimum, you should provide the following information for every instrument you create:

- **Name**. The name associated with your custom instrument in the library.
- **Category**. The category in which your instrument appears in the library. You can specify the name of an existing category—such as Memory—or create your own.
- **Description**. The instrument description, used in both the library window and in the instrument's help tag.

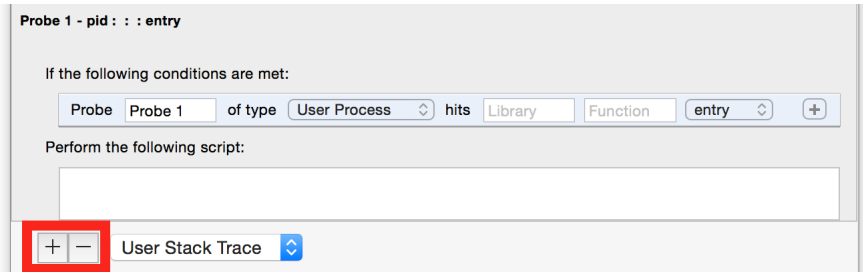
- **Probe provider**. The probe type and the details of when it should fire. Typically, this involves specifying the method or function to which the probe applies. For more information, see Specifying the Probe Provider (page 124).
- **Probe action**. The data to record or the script to execute when your probe fires; see Adding Actions to a Probe (page 129).

An instrument should contain at least one probe and may contain more than one. The probe definition consists of the provider information, predicate information, and action. All probes must specify the provider information at a minimum, and nearly all probes define some sort of action. The predicate portion of a probe definition is optional but can be a very useful tool for focusing your instrument on the correct data.

Adding and Deleting Probes

Every new instrument comes with one probe that you can configure. To add more probes, click the Add button (+) at the bottom of the instrument configuration dialog.

To remove a probe from your instrument, click the probe to select it and click the Remove button (-) at the bottom of the instrument configuration dialog.



When adding probes, it is a good idea to provide a descriptive name for the probe. By default, Instruments enumerates probes with names like Probe 1 and Probe 2.

Specifying the Probe Provider

To specify the location point or event that triggers a probe, you must associate the appropriate provider with the probe. Providers are kernel modules that act as agents for DTrace, providing the instrumentation necessary to create probes. You do not need to know how providers operate to create an instrument, but you do need to know the basic capabilities of each provider. Table 12-1 lists the providers that are supported by the Instruments app and available for use in your custom instruments. The Provider column lists the name displayed in the instrument configuration sheet, and the DTrace provider column lists the actual name of the provider used in the corresponding DTrace script.

Table 12-1 DTrace providers

Provider	DTrace provider	Description
User Process	`pid`	The probe fires on entry (or return) of the specified function in your code. You must provide the function name and the name of the library that contains it.
Objective-C	`objc`	The probe fires on entry (or return) of the specified Objective-C method. You must provide the method name and the class to which it belongs.
System Call	`syscall`	The probe fires on entry (or return) of the specified system library function.
DTrace	`DTrace`	The probe fires when DTrace itself enters a BEGIN, END, or ERROR block.
Kernel Function Boundaries	`fbt`	The probe fires on entry (or return) of the specified kernel function in your code. You must provide the kernel function name and the name of the library that contains it.
Mach	`mach_trap`	The probe fires on entry (or return) of the specified Mach library function.
Profile	`profile`	The probe fires regularly at the specified time interval on each core of the machine. Profile probes can fire with a granularity that ranges from microseconds to days.
Tick	`tick`	The probe fires at periodic intervals on one core of the machine. Tick probes can fire with a granularity that ranges from microseconds to days. You might use this provider to perform periodic tasks that are not required to be on a particular core.
I/O	`io`	The probe fires at the start of the specified kernel routine. For a list of functions monitored by this probe, use the `dtrace -l` command from Terminal to get a list of probe points. You can then search this list for probes monitored by the `io` module.
Kernel Process	`proc`	The probe fires on the initiation of one of several kernel-level routines. For a list of functions monitored by this probe, use the `dtrace -l` command from Terminal to get a list of probe points. You can then search this list for functions monitored by the `proc` module.
User-Level Synchronization	`plockstat`	The probe fires at one of several synchronization points. You can use this provider to monitor mutex and read-write lock events.

Provider	DTrace provider	Description
CPU Scheduling	`sched`	The probe fires when CPU scheduling events occur.
Core Data	`CoreData`	The probe fires at one of several Core Data–specific events. For a list of methods monitored by this probe, use the `dtrace -l` command from Terminal to get a list of probe points. You can then search this list for methods monitored by the `CoreData` module.

After selecting the provider for your probe, you need to specify the information needed by the probe. For example, for some function-level probes, providers may need function or method names, along with your code module or else the class containing your module. Other providers may only need you to select appropriate events from a pop-up menu.

After you have configured a probe, you can proceed to add additional predicates to it (to determine when it should fire) or you can go ahead and define the action for that probe.

Adding Predicates to a Probe

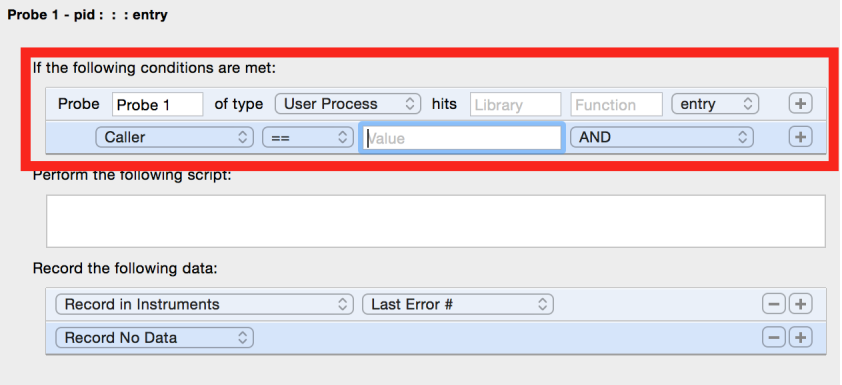
Predicates give you control over when a probe's action is executed by Instruments. You can use predicates to prevent Instruments from gathering data when you don't want it or think the data might be erroneous. For example, if your code exhibits unusual behavior only when the stack reaches a certain depth, you can use a predicate to specify the minimum target stack depth. Every time a probe fires, Instruments evaluates the associated predicates. Only if they evaluate to true does DTrace perform the associated actions.

To add a predicate to a probe

1. Click the Add button (+) in the probe conditions.
2. Select the type of predicate.

3. Define the predicate values.

Figure 12-2 Configuring a probe



You can add subsequent predicates using the Add buttons (+) of either the probe or the predicate. To remove a predicate, click the Remove button (-) next to the predicate.

Instruments evaluates predicates from top to bottom in the order in which they appear. To rearrange predicates, click the predicate's row and drag it to a new location in the table. You can link predicates using `AND` and `OR` operators, but you cannot group them to create nested condition blocks. Instead, order your predicates carefully to ensure that all of the appropriate conditions are checked.

Use the first pop-up menu in a predicate row to choose the data to inspect as part of the condition. Table 12-2 lists the standard variables defined by DTrace that you can use in your predicates or script code. The Variable column lists the name as it appears in the instrument configuration panel, and the "DTrace variable" column lists the actual name of the variable used in corresponding DTrace scripts. In addition to testing the standard variables, you can test against custom variables and constants from your script code by specifying the Custom variable type in the predicate field.

Table 12-2 DTrace variables

Variable	DTrace variable	Description
Caller	`caller`	The value of the current thread's program counter just before entering the probe. This variable contains an integer value.
Chip	`chip`	The identifier for the physical chip executing the probe. This is a 0-based integer indicating the index of the current core. For example, a four-core machine has cores 0 through 3.

Variable	DTrace variable	Description
CPU	`cpu`	The identifier for the CPU executing the probe. This is a 0-based integer indicating the index of the current core. For example, a four-core machine has cores 0 through 3.
Current Working Directory	`cwd`	The current working directory of the current process. This variable contains a string value.
Last Error #	`errno`	The error value returned by the last system call made on the current thread. This variable contains an integer value.
Executable	`execname`	The name that was passed to `exec` to execute the current process. This variable contains a string value.
User ID	`uid`	The real user ID of the current process. This variable contains an integer value.
Group ID	`gid`	The real group ID of the current process. This variable contains an integer value.
Process ID	`pid`	The process ID of the current process. This variable contains an integer value.
Parent ID	`ppid`	The process ID of the parent process. This variable contains an integer value.
Thread ID	`tid`	The thread ID of the current thread. This is the same value returned by the `pthread_self` function.
Interrupt Priority Level	`ipl`	The interrupt priority level on the current CPU at the time the probe fired. This variable contains an unsigned integer value.
Function	`probefunc`	The function name part of the probe's description. This variable contains a string value.
Module	`probemod`	The module name part of the probe's description. This variable contains a string value.
Name	`probename`	The name portion of the probe's description. This variable contains a string value.

Variable	DTrace variable	Description
Provider	`probeprov`	The provider name part of the probe's description. This variable contains a string value.
Root Directory	`root`	The root directory of the process. This variable contains a string value.
Stack Depth	`stackdepth`	The stack frame depth of the current thread at the time the thread fired. This variable contains an unsigned integer value.
Relative Timestamp	`timestamp`	The current value of the system's timestamp counter, in nanoseconds. Because this counter increments from an arbitrary point in the past, use it to calculate only relative time differences. This variable contains an unsigned 64-bit integer value.
Virtual Timestamp	`vtimestamp`	The amount of time the current thread has been running, in nanoseconds. This value does not include time spent in DTrace predicates and actions. This variable contains an unsigned 64-bit integer value.
Timestamp	`walltimestamp/1000`	The current number of nanoseconds that have elapsed since 00:00 Universal coordinated Time, January 1, 1970. This variable contains an unsigned 64-bit integer value.
`arg0` through `arg9`	`arg0` through `arg9`	The first 10 arguments to the probe, represented as raw 64-bit integers. If fewer than ten arguments were passed to the probe, the remaining variables contain the value 0.
Custom	The name of your variable	Use this option to specify a variable or constant from one of your scripts.

In addition to specifying the condition variable, you must specify the comparison operator and the target value.

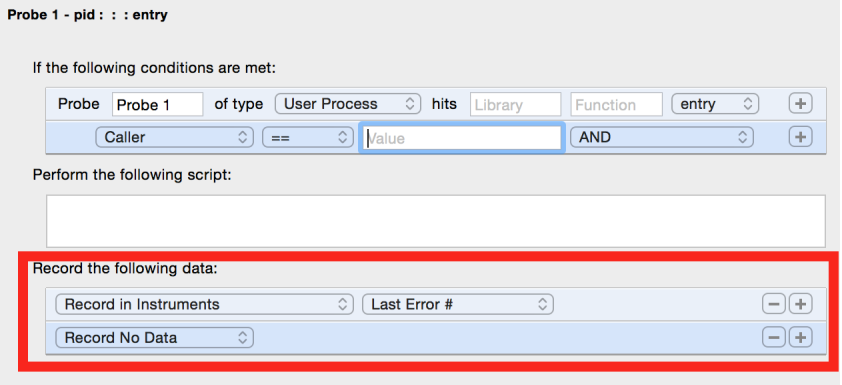
Adding Actions to a Probe

When a probe point defined by your instrument is hit and the probe's predicate conditions evaluate to true, DTrace runs the actions associated with the probe. You use your probe's actions to gather data or to perform some additional processing. For example, if your probe monitors a specific function or method, you could have it return the caller of that function and any stack trace information to Instruments. If you wanted a slightly more advanced action, you could use a script variable to track the number of times the function was called

and report that information as well. And if you wanted an even more advanced action, you could write a script that uses kernel-level DTrace functions to determine the status of a lock used by your function. In this latter case, your script code might also return the current owner of the lock (if there is one) to help you determine the interactions among your code's different threads.

Figure 12-3 shows the portion of the instrument configuration sheet where you specify your probe's actions. The script portion simply contains a text field for you to type in your script code. (Instruments does not validate your code before passing it to DTrace, so check your code carefully.) The bottom section contains controls for specifying the data you want DTrace to return to Instruments. You can use the pop-up menus to configure the built-in DTrace variables you want to return. You can also select Custom from this pop-up menu and return one of your script variables.

Figure 12-3 Configuring a probe's action



When you configure your instrument to return a custom variable, Instruments asks you to provide the following information:

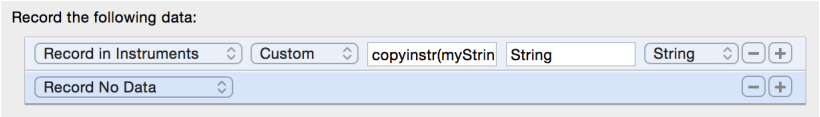
- The script variable containing the data
- The name to apply to the variable in your instrument interface
- The type of the variable

Any data your probe returned to Instruments is collected and displayed in your instrument's detail pane. The detail pane displays all data variables regardless of type. If stack trace information is available for a specific probe, Instruments displays that information in your instrument's Extended Detail inspector. In addition, Instruments automatically looks for integer data types returned by your instrument and adds those types to the list of statistics your instrument can display in the track pane.

Because DTrace scripts run in kernel space and the Instruments app runs in user space, if you want to return the value of a custom pointer-based script variable to Instruments, you must create a buffer to hold the variable's data. The simplest way to create a buffer is to use the `copyin` or `copyinstr` subroutines found in DTrace.

The `copyinstr` subroutine takes a pointer to a C string and returns the contents of the string in a form you can return to Instruments. Similarly, the `copyin` subroutine takes a pointer and size value and returns a buffer to the data, which you can later format into a string using the `stringof` keyword. Both of these subroutines are part of the DTrace environment and can be used from any part of your probe's action definition. For example, to return the string from a C-style string pointer, you simply wrap the variable name with the `copyinstr` subroutine, as shown in Figure 12-4.

Figure 12-4 Returning a string pointer



Important: Instruments automatically wraps built-in variables (such as the `arg0` through `arg9` function arguments) with a call to `copyinstr` if the variable type is set to `string`. Instruments does not automatically wrap script's custom variables, however. You are responsible for ensuring that the data in a custom variable actually matches the type specified for that variable.

For a list of the built-in variables supported by Instruments, see Table 12-2 (page 127). For more information on scripts and script variables, see Tips for Writing Custom Scripts (page 131). For more information on DTrace subroutines, including the `copyin` and `copyinstr` subroutines, see the *Solaris Dynamic Tracing Guide*, available from the Oracle Technology Network.

Tips for Writing Custom Scripts

You write DTrace scripts using the D scripting language, whose syntax is derived from a large subset of the C programming language. The D language combines the programming constructs of the C language with a special set of functions and variables to help you trace information in your app.

The following sections describe some of the common ways to use scripts in your custom instruments. These sections do not provide a comprehensive overview of the D language or the process for writing DTrace scripts. For information about scripting and the D language, see the *Solaris Dynamic Tracing Guide*, available from the Oracle Technology Network.

Writing BEGIN and END Scripts

If you want to do more than return the information in DTrace's built-in variables to Instruments whenever your action fires, you need to write custom scripts. Scripts interact directly with DTrace at the kernel level, providing access to low-level information about the kernel and the active process. Most instruments use scripts to gather

information not readily available from DTrace. You can also use scripts to manipulate raw data before returning it to Instruments. For example, you can use a script to normalize a data value to a specific range if you want to make it easier to compare that value graphically with other values in your instrument's track pane.

In Instruments, the custom instrument configuration sheet provides several areas where you can write DTrace scripts:

- The DATA section contains definitions of any global variables you want to use in your instrument.
- The BEGIN section contains any initialization code for your instrument.
- Each probe contains script code as part of its action.
- The END section contains any clean up code for your instrument.

All script sections are optional. You are not required to have initialization scripts or cleanup scripts if your instrument does not need them. If your instrument defines global variables in its DATA section, however, it is recommended that you also provide an initialization script to set those variables to a known value. The D language does not allow you to assign values inline with your global variable declarations, so you must put those assignments in your BEGIN section. For example, a simple DATA section might consist of a single variable declaration, such as the following:

```
int myVariable;
```

The corresponding BEGIN section would then contain the following code to initialize that variable:

```
myVariable = 0;
```

If your corresponding probe actions change the value of `myVariable`, you might use the END section of your probe to format and print out the final value of the variable.

Most of your script code is likely to be associated with individual probes. Each probe can have a script associated with its action. When it comes time to execute a probe's action, DTrace runs your script code first and then returns any requested data back to Instruments. Because passing data back to Instruments involves copying data from the kernel space back to the Instruments app space, you should always pass data back to Instruments by configuring the appropriate entries in the "Record the following data:" section of the instrument configuration sheet. Variables returned manually from your script code may not be returned correctly to Instruments.

Accessing Kernel Data from Custom Scripts

Because DTrace scripts execute inside the system kernel, they have access to kernel symbols. If you want to look at global kernel variables and data structures from your custom instruments, you can do so in your DTrace scripts. To access a kernel variable, precede the name of the variable with the backquote character (`). The backquote character tells DTrace to look for the specified variable outside of the current script.

Listing 12-1 shows a sample action script that retrieves the current load information from the `avenrun` kernel variable and uses that variable to calculate a one-minute average load of the system. If you were to create a probe using the Profile provider, you could have this script gather load data periodically and then graph that information in Instruments.

Listing 12-1 Accessing kernel variables from a DTrace script

```
this->load1a = `avenrun[0]/1000;
this->load1b = ((`avenrun[0] % 1000) * 100) / 1000;
this->load1 = (100 * this->load1a) + this->load1b;
```

Scoping Variables Appropriately

DTrace scripts have an essentially flat structure, due to a lack of flow control statements and the desire to keep probe execution time to a minimum. That said, you can scope the variables in DTrace scripts to different levels depending on your need. Table 12-3 lists the scoping levels for variables and the syntax for using variables at each level.

Table 12-3 Variable scope in DTrace scripts

Scope	Syntax example	Description
Global	`myGlobal = 1;`	Global variables are identified simply using the variable name. All probe actions on all system threads have access to variables in this space.
Thread	`self->myThreadVar = 1;`	Thread-local variables are dereferenced from the `self` keyword. All probe actions running on the same thread have access to variables in this space. You might use this scope to collect data over the course of several runs of a probe's action on the current thread.
Probe	`this->myLocalVar = 1;`	Probe-local variables are dereferenced using the `this` keyword. Only the current running probe has access to variables in this space. Typically, you use this scope to define temporary variables that you want the kernel to clean up when the current action ends.

Finding Script Errors

If the script code for one of your custom instruments contains an error, Instruments displays an error message in the track pane when DTrace compiles the script. Instruments reports the error after you press the Record button in your trace document but before tracing actually begins. Inside the error message bubble is an Edit button. Clicking this button opens the instrument configuration sheet, which now identifies the probe with the error.

Exporting DTrace Scripts

Although Instruments provides a convenient interface for gathering trace data, there are still times when it is more convenient to gather trace data directly using DTrace. If you are a system administrator or are writing automated test scripts, for example, you might prefer to use the DTrace command-line interface to launch a process and gather the data. Using the command-line tool requires you to write your own DTrace scripts, which can be time consuming and can lead to errors. If you already have a trace document with one or more DTrace-based instruments, you can use the Instruments app to generate a DTrace script that provides the same behavior as the instruments in your trace document.

Instruments supports exporting DTrace scripts only for documents where all of the instruments are based on DTrace. This means that your document can include custom instruments and a handful of the built-in instruments, such as the instruments in the File System and CoreData groups in the Library window.

To export a DTrace script

1. Select the trace document.
2. Click File > DTrace Script Export.
3. Enter a name for the DTrace script.
4. Select a location for the DTrace script.
5. Click Save.

The DTrace Script Export command places the script commands for your instruments in a text file that you can then pass to the `dtrace` command-line tool using the `-s` option. For example, if you export a script named `MyInstrumentsScript.d`, run it from Terminal using the following command:

```
sudo dtrace -s MyInstrumentsScript.d
```

> **Note:** You must have superuser privileges to run `dtrace` in most instances, which is why the `sudo` command is used to run `dtrace` in the preceding example.

Another advantage of exporting your scripts from Instruments (as opposed to writing them manually) is that after running the script, you can import the resulting data back into Instruments and review it there. Scripts exported from Instruments print a start marker (with the text `dtrace_output_begin`) at the beginning of the DTrace output. To gather the data, simply copy all of the DTrace output (including the start marker) from Terminal and paste it into a text file, or just redirect the output from the `dtrace` tool directly to a file. To import the data in Instruments, select the trace document from which you generated the original script and choose File > DTrace Data Import.

Preferences

The Preferences window is accessed by selecting Instruments > Preferences. It contains six tabs where you can customize Instruments to best suit your needs.

General Tab

Use the General tab (see The general preference pane in Instruments) to configure basic Instruments preferences including startup, keyboard shortcuts, and warnings options.

Table A-1 General tab

Option	Description
Always use deferred mode	Performs data analysis for all traces after data collection is complete.
Automatically time profile spinning applications	Automatically monitors for a spinning process while a trace is recorded. This can be a process other than the one being recorded. If detected, Instruments starts the Time Profiler instrument on the spinning process.
Suppress template chooser	Hides the template chooser when Instruments starts up and when a new trace document is created.
Save current run only	Saves only the current data collection run for each individual instrument.
Compress run data	Compresses each saved run into zip format.
Default document location	Specifies the location where new Instruments documents are created. By default, this is a temporary directory. Select Choose from this pop-up menu to use a different directory. Click Reset to use the temporary directory again.
Open Keyboard Shortcut Preferences	Opens the Keyboard > Shortcuts > Services pane in the System Preferences app, as shown in The Keyboard > Shortcuts > Services pane in System Preferences. From here, you can assign keyboard shortcuts to development services, such as a service that automatically opens an Xcode project in Instruments and profiles it with the System Trace template.

Option	Description
Reset “Don’t Ask Me” Warnings	Reenables dialog warnings you previously elected not to show. Instruments has several warning dialogs that you can disable by selecting the “Do not show this message again” checkbox in the dialog. To reenable all of these warning dialogs, click the Reset “Don’t Ask Me” Warnings button.

Figure A-1 The General preference pane in Instruments

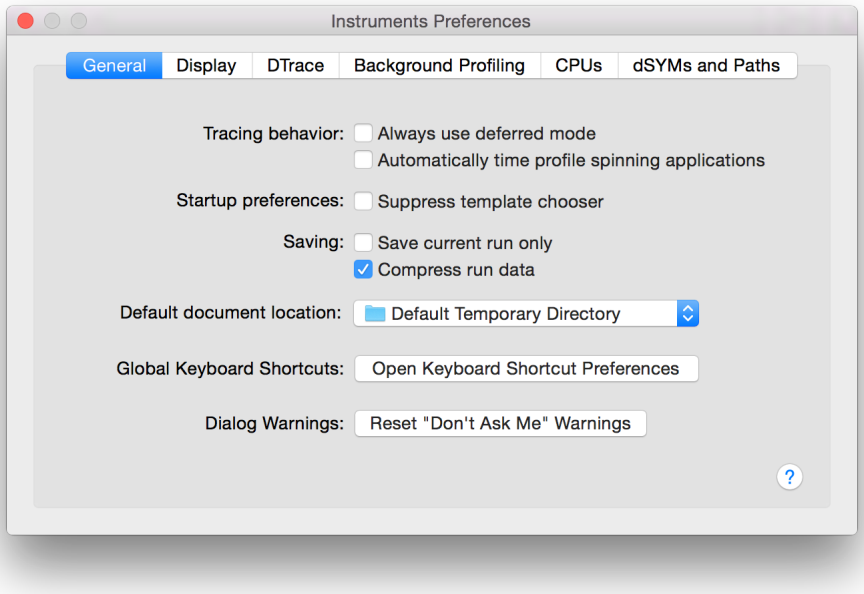
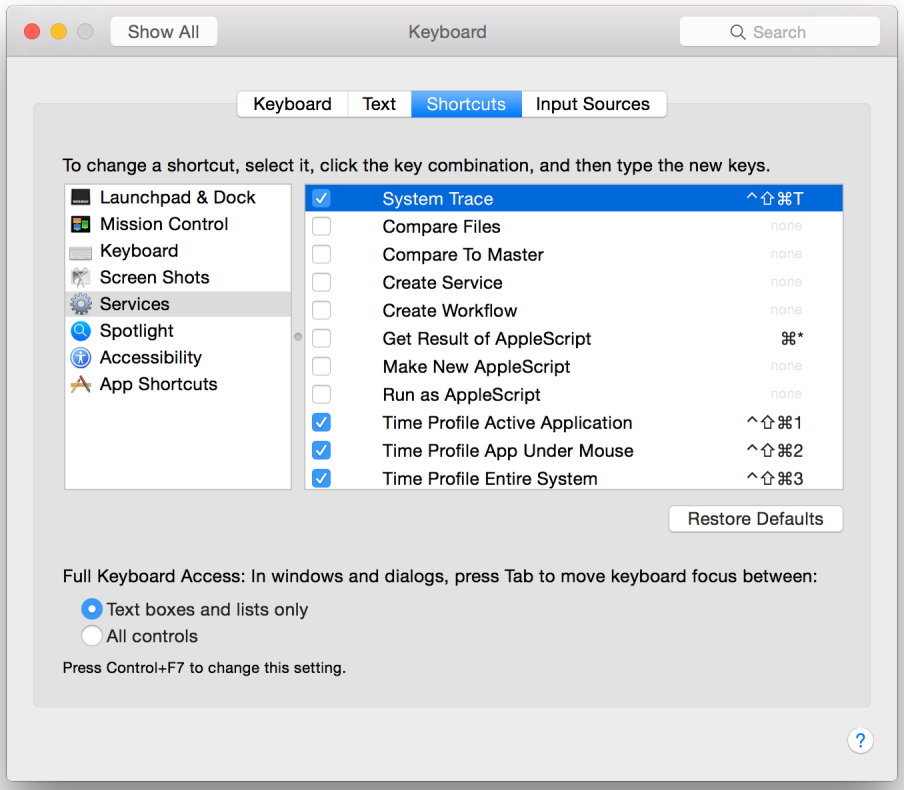


Figure A-2 The Keyboard > Shortcuts > Services pane in System Preferences



Display Tab

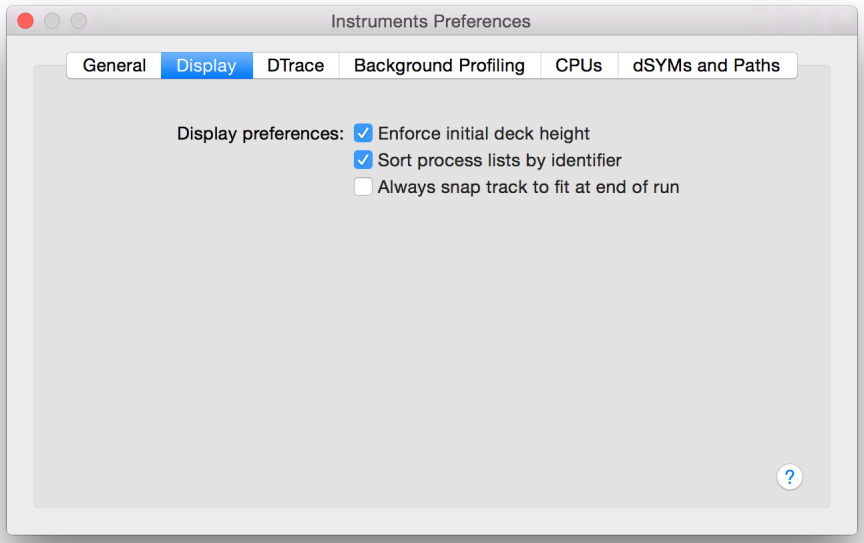
Use the Display tab (see The Display preference pane in Instruments) to configure track display options in a trace document.

Table A-2 Display tab

Option	Description
Enforce initial deck height	When selected, prevents custom deck height from being restored when an instrument document is re-loaded and uses the template's default deck height. When deselected, saves and restores the current deck height.
Sort process lists by identifier	When selected, sorts all process lists, such as the attach menu, by their process ID. When deselected, sorts process lists alphabetically.

Option	Description
Always snap track to fit at end of run	Automatically scales the track in a trace document at the end of a run to fit all data in the window.

Figure A-3 The Display preference pane in Instruments



DTrace Tab

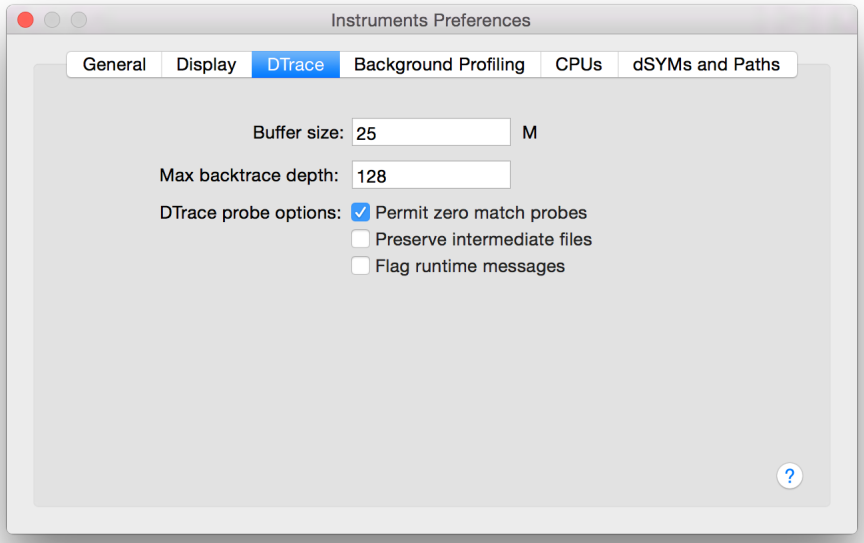
Use the DTrace tab (see The DTrace preference pane in Instruments) to configure how DTrace-based instruments act. DTrace instruments use dynamic tracing to access low-level kernel operations and user processes running on your device.

Table A-3 DTrace tab

Option	Description
Buffer size	Sets the size of the DTrace kernel buffer (in megabytes). The default is 25 MB.
Max backtrace depth	Sets the maximum stack depth that is captured when using a DTrace instrument. The default is 128.
Permit zero match probes	Prevents an error when a specified probe is not found.

Option	Description
Preserve intermediate files	Prevents Instruments from removing intermediate DTrace data output files from the disk.
Flag runtime messages	Adds flags to the timeline for DTrace runtime status and error messages encountered during a recording.

Figure A-4 The DTrace preference pane in Instruments



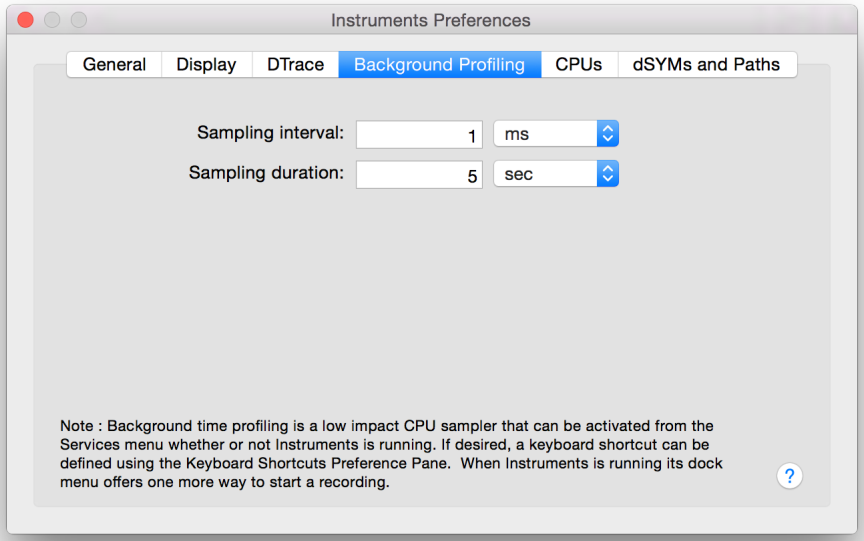
Background Profiling Tab

Use the Background Profiling tab (see The Background Profiling preference pane in Instruments) to configure how the Time Profiler instrument behaves when operating in the background. Since background time profiling is a low impact CPU sampler, you can activate it without Instruments running. To do this, enable the time profiling services in the Keyboard > Shortcuts > Services pane in System Preferences. You can even assign keyboard shortcuts to these services, as shown in The Keyboard > Shortcuts > Services pane in System Preferences. When Instruments is running, time profiling can be started from the Instruments Dock menu (Control+Click on the Instruments icon in the dock to display this menu)

Table A-4 Background Profiling tab

Option	Description
Sampling interval	Specifies how often a sample is taken. Type a numeric value in the field. Choose microsecond (µs), millisecond (ms), or second (sec) from the pop-up menu. Defaults to 1 millisecond.
Sampling duration	Sets the length of a sample trace. Type a numeric value in the field. Choose microsecond (µs), millisecond (ms), or second (sec) from the pop-up menu. Defaults to 5 seconds.

Figure A-5 The Background Profiling preference pane in Instruments



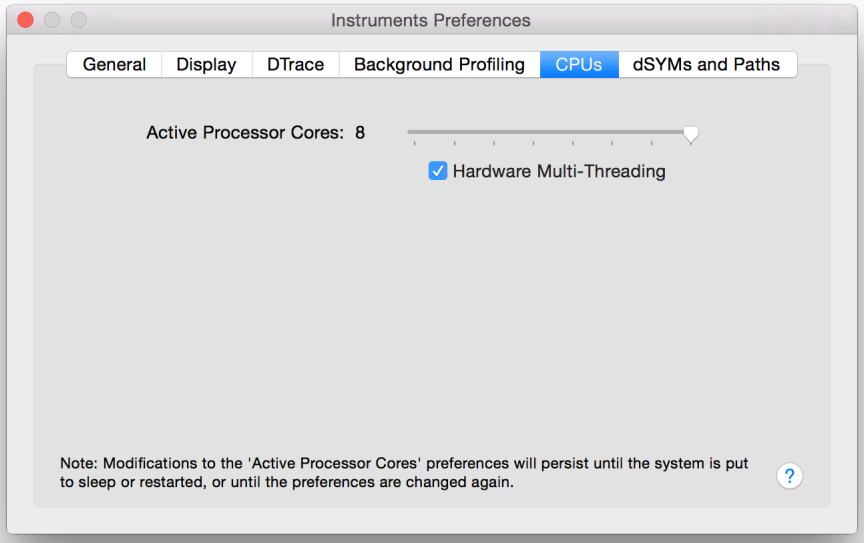
CPUs Tab

Use the CPUs tab (see The CPUs preference pane in Instruments) to configure Instruments for the CPU configuration of your device.

Table A-5 CPUs tab

Option	Description
Active Processor Cores	Adjusts how many cores of your system are active. Only active cores are scheduled to perform any operations when profiling. Use the slider to set the number of active cores equal to the number of cores on the device that you expect your application to run on. Changes to this preference persist until you change it again, or until your system is put to sleep or restarted.
Hardware Multi-Threading	Allows CPU cores to utilize multiple execution units. When disabled, there is only one active execution unit per processor core.

Figure A-6 The CPUs preference pane in Instruments



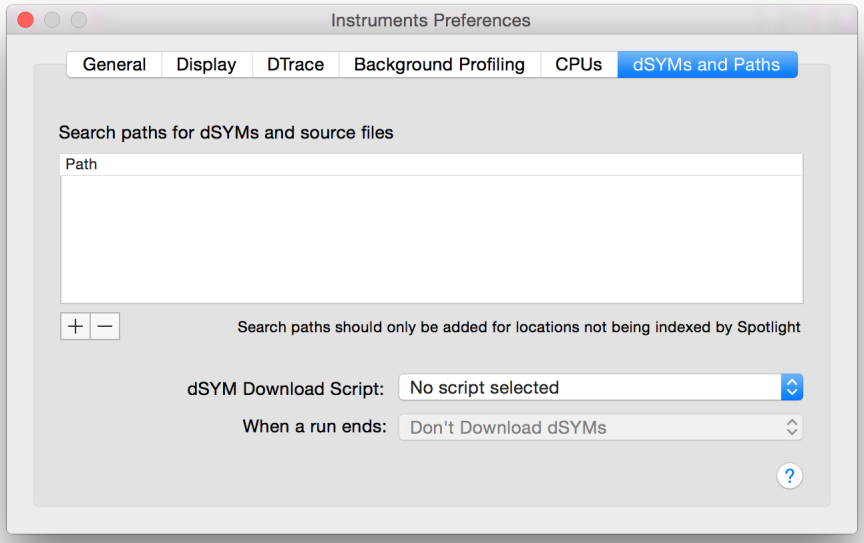
dSYMs and Paths Tab

Use the “dSYMs and Paths” tab (see The dSYMs and Paths preference pane in Instruments) to set global search paths for Instruments.

Table A-6 "dSYMs and Paths tab"

Option	Description
+	Adds a search location. Opens the Open Directory dialog. Navigate to the desired directory and click Open. Directories indexed by Spotlight are already searched, so you only need to add search paths for directories that aren't indexed.
–	Removes a selected search location. Select a path and click the minus (-) button to remove the path.
dSYM Download Script	Provides the option of running a custom script to locate and access the necessary dSYM files. This option is provided for use by large developers with distributed code databases; it is not needed by the majority of developers.
When a run ends	Enabled only if a dSYM download script is specified. Configures how the download script is automatically used after a recording has finished. Options include "Don't Download dSYMs," "Download App dSYMs," "Download App and User Framework dSYMs," and "Download All dSYMs." The default is "Don't Download dSYMs."

Figure A-7 The "dSYMs and Paths" preference pane in Instruments



Keyboard Shortcuts

Keyboard shortcuts provide an easy way for experienced users to perform actions without a mouse click. This appendix provides a list of keyboard shortcuts provided by the Instruments app.

Instruments Menu

Here are keyboard shortcuts for the Instruments menu:

Table B-1 Instruments menu keyboard shortcuts

Keys	Commands
Command-Comma (,)	Preferences
Command-H	Hide Instruments
Command-Option-H	Hide Others
Command-Q	Quit Instruments

File Menu

Here are keyboard shortcuts for the File menu:

Table B-2 File menu keyboard shortcuts

Keys	Commands
Command-N	New
Command-O	Open
Command-W	Close
Command-S	Save
Command-Shift-S	Save as

Keys	Commands
Command-R	Record Trace
Command-Shift-R	Pause Trace
Command-Option-R	Record Options

Edit Menu

Here are the keyboard shortcuts for the Edit menu:

Table B-3 Edit menu keyboard shortcuts

Keys	Commands
Command-Z	Undo
Command-Shift-Z	Redo
Command-X	Cut
Command-C	Copy
Command-Shift-C	Deep Copy
Command-V	Paste
Shift-Option-Command-V	Paste and Match Style
Command-A	Select All
Command-F	Find
Command-G	Find Next
Command-Shift-G	Find Previous
Command-E	Use Selection for Find
Command-J	Jump to Selection
Command-Colon (:)	Show Spelling and Grammar
Command-Semicolon (;)	Check Spelling
Command-Down Arrow	Add Flag

Keys	Commands
Command-Up Arrow	Remove Flag
Command-Control-Space	Special Characters

View Menu

Here are the keyboard shortcuts for the View menu:

Table B-4 View menu keyboard shortcuts

Keys	Commands
Command-D	Detail
Command-1	Show Record Settings
Command-2	Show Display Settings
Command-3	Show Extended Detail
Command-Control-F	Full Screen
Command-Right Arrow	Next Flag
Command-Left Arrow	Previous Flag
Command-Dash (-)	Decrease Deck Size
Command-Plus (+)	Increase Deck Size
Command-Control-Z	Snap Track To Fit
Command-Less Than (<)	Set Inspection Range End
Command-Greater Than (>)	Set Inspection Range End
Command-Period (.)	Clear Inspection Range

Instrument Menu

Here are the keyboard shortcuts for the Instrument menu:

Table B-5 Instrument menu keyboard shortcuts

Keys	Commands
Command-B	Build New Instrument
Command-T	Trace Symbol
Command-Quote (")	Previous Run
Command-Apostrophe (')	Next Run

Window Menu

Here are the keyboard shortcuts for the Window menu:

Table B-6 Window menu keyboard shortcuts

Keys	Commands
Command-M	Minimize
Control-Z	Zoom
Command-L	Library
Command-Shift-T	Manage Flags

Menu Bar Definitions

The menu bar contains a collection of menus allowing you to control Instruments and manipulate recorded data. Each menu contains a list of commands that have been grouped by function.

Instruments Menu

The Instruments menu provides commands for displaying the Instruments window.

Table C-1 Instruments menu

Command	Description
About Instruments	Provides version and copyright information for Instruments.
Preferences	Opens the Preferences window.
Services	Allows you to trigger Instrument-related services and services from other apps, if enabled. Select Services Preferences from this menu to open the Keyboard > Shortcuts > Services pane in System Preferences, where you can enable or disable services, and assign keyboard shortcuts to services.
Hide Instruments	Hides all Instruments windows and menus.
Hide Others	Hides all processes except for Instruments.
Show All	Displays all hidden processes.
Quit Instruments	Closes all trace documents and shuts down Instruments. You are asked whether you want to save any unsaved data.

File Menu

The File menu provides commands for creating, importing, saving, and running trace documents.

Table C-2 File menu

Command	Description
New	Creates a new trace document.
Open	Opens a specified saved trace document.
Open Recent	Opens a recently saved trace document. Select Clear Menu from this menu to remove all recently saved trace documents from the list.
Import Data	Opens the Import Trace dialog allowing you to load previously saved data. When loading data, you can choose an instrument to import the data. Options include Activity Monitor, Allocations, Counters, Leaks, Sampler, Time Profiler, Event Profiler, and VM Tracker.
Close	Closes the selected trace document. You are asked whether you want to save any unsaved data.
Save	Saves the selected trace document. Asks for a name and destination if this is the first time the trace document has been saved.
Save As	Saves the selected trace document in a designated location with a specified name. Gives you the option to save the current run only.
Save As Template	Saves the current collection of instruments as a trace template. Trace templates can be accessed from the Instruments starting dialog. By default, templates are saved in `~/Library/Application Support/Instruments/Templates`.
Record Trace	Records trace data using the current instruments. While recording, this menu changes to Stop Trace, and allows you to stop the currently running trace.
Pause Trace	Pauses the currently running trace.
Loop	Determines if the user interface recorder will loop during play back to repeat the recorded steps continuously.
Record Options	Opens the Record Options dialog. Allows you to set the start delay, time limit, window limit, and whether data is collected in deferred mode.
DTrace Data Import	Allows you to import previously collected DTrace data.
DTrace Script Export	Allows you to export DTrace scripts.
Import Logged Data from Device	Allows you to import logs from an iOS device.

Command	Description
Symbols	Opens the symbolication dialog. Allows you to set the paths and files used when looking for symbols.

Edit Menu

The Edit menu provides commands for text manipulation, including spelling, grammar, and search capabilities. You can also re-symbolicate a trace document.

Table C-3 Edit menu

Command	Description
Undo	Removes previously typed instructions.
Redo	Applies instructions previously removed by Undo.
Cut	Removes the selected text from the document.
Copy	Copies the selected item to the Clipboard.
Deep Copy	Copies the selected item and all of the items contained in it on to the Clipboard.
Paste	Pastes the current information stored on the Clipboard into the selected document.
Paste and Match Style	Pastes the current information stored on the Clipboard into the selected document and uses the document's style information for the data.
Delete	Deletes the selected item.
Select All	Selects all available items.

Command	Description
Find	Allows you to search for an item. Contains five suboptions: • Find: Opens the search field and looks for the first instance of the input text. • Find Next: Opens the search field and looks for the next instance of the input text. • Find Previous: Opens the search field and looks for the previous instance of the input text. • Use Selection for Find: Opens the search field and uses the highlighted selection as the input. • Jump to Selection: Jumps to the current selection.
Spelling	Provides spelling and grammar options for the document. Contains three suboptions: • Show Spelling and Grammar: Opens the Spelling and Grammar dialog. • Check Spelling: Performs a spell check on the document. • Check Spelling While Typing: Performs a spell check while typing in the document.
Add Flag	Adds a new flag to the track pane.
Remove Flag	Removes an existing flag from the track pane.
Start Dictation	Allows you to dictate text content into a selected text field.
Special Characters	Opens the Characters palette, which allows you to insert characters such as bullets, emoji, math symbols, and more.

View Menu

The View menu provides commands for manipulating the Instruments window.

Table C-4 View menu

Command	Description
Detail	Opens and closes the Detail pane.

Command	Description
Inspectors	Shows different inspectors in the Detail pane. Options include showing the Record Settings inspector, the Display Settings inspector, or the Extended Detail inspector.
Full Screen	Toggles Instruments in and out of full screen mode.
Mini Instruments	Toggles Instruments in and out of mini mode.
Next Flag	Jumps to the next flag in the track pane.
Previous Flag	Jumps to the previous flag in the track pane.
Decrease Deck Size	Decreases the height of the selected instrument.
Increase Deck Size	Increases the height of the selected instrument.
Snap Track To Fit	Resizes collected data so that it fits exactly inside of the track pane.
Set Inspection Range Start	Sets the start of the inspection range to the current position in the track pane.
Set Inspection Range End	Sets the end of the inspection range to the current position in the track pane.
Clear Inspection Range	Clears the current inspection range in the track pane.

Instrument Menu

The Instrument menu provides commands for collecting and recording trace data.

Table C-5 Instrument menu

Command	Description
Build New Instrument	Opens the Build Instrument dialog.
Edit <Instrument Name> Instrument	Opens the Build Instrument dialog filled with the selected instrument's information.
Delete <Instrument Name> Instrument	After confirmation, removes the selected instrument from the trace document.

Command	Description
Delete Run <#>	After confirmation, deletes the selected data collection run from the trace document.
Trace Symbol	Brings up the trace symbol dialog. Allows you to trace a specific symbol.
Previous Run	Displays the run prior to the currently displayed run, if one exists.
Next Run	Displays the run after the currently displayed run, if one exists.
Export Track for <Instrument Name - Process>	Saves the collected information for the instrument / process combination listed.
Compare Call Trees	Compares the call tree of the current run with the call trees of previous runs.
Call Tree Data Mining	Allows you to manipulate the call tree in the Detail pane by .

Window Menu

The Window menu provides commands resizing the Instruments window along with managing flags and PM events.

Table C-6 Window menu

Command	Description
Minimize	Minimizes the Instruments window to the dock.
Zoom	Enlarges the Instruments window to the size of your video monitor while leaving the menu bar available.
Library	Opens and closes the Library pane. This pane displays the list of instruments, which you can drag and drop into your Instruments document.
Manage Flags	Opens the Flags dialog, where you can edit flag information.
Bring All to Front	Brings all Instruments windows to the front of the screen.
<WindowName>	Makes the corresponding window active.

Trace Template Contents

The appendix provides you with a quick way to view the instruments contained within the individual trace templates.

Trace Templates

Contains a list of pre-built profiling templates for performing common traces.

Trace template	Instruments
Activity Monitor	Activity Monitor Instrument
Allocations	Allocations Instrument VM Tracker Instrument
Automation	Automation Instrument
Cocoa Layout	Cocoa Layout Instrument
Core Animation	Core Animation Instrument Time Profiler Instrument
Core Data	Core Data Cache Misses Instrument Core Data Fetches Instrument Core Data Saves Instrument
Counters	Counters Instrument
Dispatch	Dispatch Instrument

Trace template	Instruments
Energy Diagnostics	Bluetooth Instrument CPU Activity Instrument Display Brightness Instrument Energy Usage Instrument GPS Instrument Network Activity Instrument Sleep/Wake Instrument WiFi Instrument
File Activity	File Activity Instrument File Attributes Instrument Directory I/O Instrument Reads/Writes Instrument
GPU Driver	GPU Driver Instrument Time Profiler Instrument
Leaks	Allocations Instrument Leaks Instrument
Multicore	Dispatch Instrument Thread States Instrument
Network	Connections Instrument
OpenGL ES Analysis	GPU Driver Instrument OpenGL ES Analyzer Instrument
Sudden Termination	Sudden Termination Instrument
System Trace	Scheduling Instrument System Calls Instrument VM Operations Instrument
System Usage	I/O Activity Instrument
Time Profiler	Time Profiler Instrument
UI Recorder	User Interface Instrument

Trace template	Instruments
Zombies	Allocations Instrument

Document Revision History

This table describes the changes to *Instruments User Guide*.

Date	Notes
2014-10-20	Expanded section on abandoned memory.
2014-10-16	Updated for Xcode 6.1
2014-09-17	Updated screenshots and notes for new Instruments 6 UI.
2014-03-10	Updated to improve and clarify description of Zombie templates. Updated to correspond to changes in Preferences UI.
2013-09-18	Updated description of automation instrument to note that it only works for apps signed with a development profile.
2013-04-23	Updated outdated graphics.
2013-01-28	Added information on the iprofiler Terminal command.
2012-09-19	Added a chapter on using the Automation instrument.
2012-06-11	Updated the document for version 4.3 and based the document on tasks.
2012-03-02	Updated information on finding Instruments and Xcode.
2011-05-07	Added information about the OpenGL ES Analyzer instrument.
2010-11-15	Updated to add information about using the Automation instrument.
2010-09-01	Updated to describe new features.
2010-07-09	Changed occurrences of “iPhone OS” to “iOS”.

Date	Notes
2010-05-27	Updated with information about new iPhone instruments.
2010-01-20	Added information about iPhone-specific instruments.
2009-08-20	Added information about the Dispatch instrument.
2009-07-24	Added information about gathering performance data from a wireless device.
2008-10-15	Made minor editorial corrections.
2008-02-08	Explained how playing protected content affects systemwide data collection.
2007-10-31	New document that describes how to use the Instruments application.

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010