Instruments User Guide

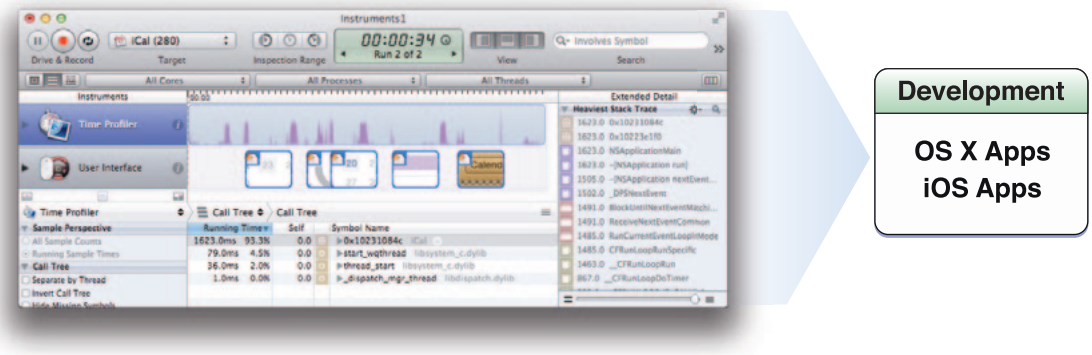
Developer

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 one or more processes 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 and replay them, using one or more instruments to gather data.

Instruments includes the ability to:

- Examine the behavior of one or more processes
- Record a sequence of user actions and replay them, reliably reproducing those events and collecting data over multiple runs
- Create your own custom DTrace instruments to analyze aspects of system and app behavior
- Save user interface recordings and instrument configurations as templates, accessible from Xcode

Using Instruments, you can:

- Track down difficult-to-reproduce problems in your code
- Do performance analysis on 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 the Instruments app:

- "Instruments Quick Start" (page 10) describes how to install Instruments and provides a quick example.
- "Touring Instruments" (page 17) 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 33) describes the ways you can initiate traces and gather trace data.
- "Examining Your Collected Data" (page 44) describes the tools you use to view the data returned by the instruments.
- "Saving and Importing Trace Data" (page 53) describes how you save trace documents and data and how you import data from other sources.
- "Locating Memory Issues in Your App" (page 56) provides examples of how to use the memory-oriented trace templates.
- "Measuring I/O Activity in iOS Devices" (page 62) provides examples of how to use the I/O-oriented trace templates.
- "Measuring Graphics Performance in Your iOS Device" (page 66) provides examples of how to use the OpenGL ES–oriented trace templates.
- "Analyzing CPU Usage in Your App" (page 70) provides examples of how to use the CPU-oriented trace templates.
- "Monitoring OS X App Activity" (page 77) provides examples of how to use the behavior-oriented trace templates.
- "Automating UI Testing" (page 81) provides examples of how to write scripts for automatic testing of your app.
- "Creating Custom Instruments" (page 101) shows how to create and configure your own DTrace-based custom instrument.

See Also

Instruments is best used in conjunction with Xcode. For information on how to use Xcode, see the *Xcode Overview*. For information on how to configure an instrument and what information it collects, see the *Instruments User Reference*.

Instruments Quick Start

Instruments is a powerful tool you can use to collect data about the performance and behavior of one or more processes 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 spot trends that might otherwise be hard to spot with other tools. For example, your app may have large memory growth that is caused by multiple network connections being opened. Using the Allocations and Connections instruments together, you can pinpoint those connections that are not closing and so are causing 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 use as is to examine various aspects of your code. You can configure instruments to gather data about the same process or about different processes on the system. You can build new instruments using the custom instrument builder interface, which uses the DTrace program to gather the data you want.

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

All work in Instruments is done in a trace document, which contains a set of instruments and the data they collected. Each trace document typically holds a single session's worth of data, which is also referred to as a *single trace*. You can save a trace document to preserve the trace data you have gathered and open them again later to view that data.

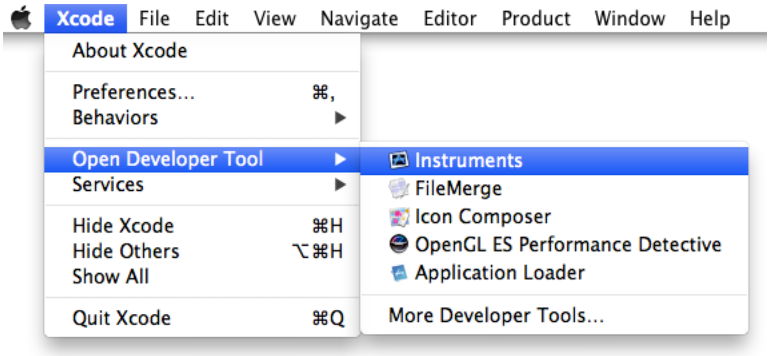
Although most instruments are geared toward *gathering* trace data, the User Interface instrument helps *automate* data collection. With it you can record user events while you gather your trace data. You can use this recording to reliably reproduce the same sequence of events over and over again. Each time you run through this sequence, your trace document gathers a new set of trace data from the other instruments in the document and presents that data side by side with previous runs. This feature lets you compare trace data as you make improvements to your code and verify that the changes you make are having the desired impact.

Launching Instruments

Instruments is contained within the Xcode 4 toolset. Download Xcode from the App Store and install it onto your computer. After you have installed Xcode, you are ready to run Instruments. Instruments can be launched in one of three ways.

To run Instruments from Xcode

1. Open Xcode.
2. Choose Xcode > Open Developer Tool > Instruments.

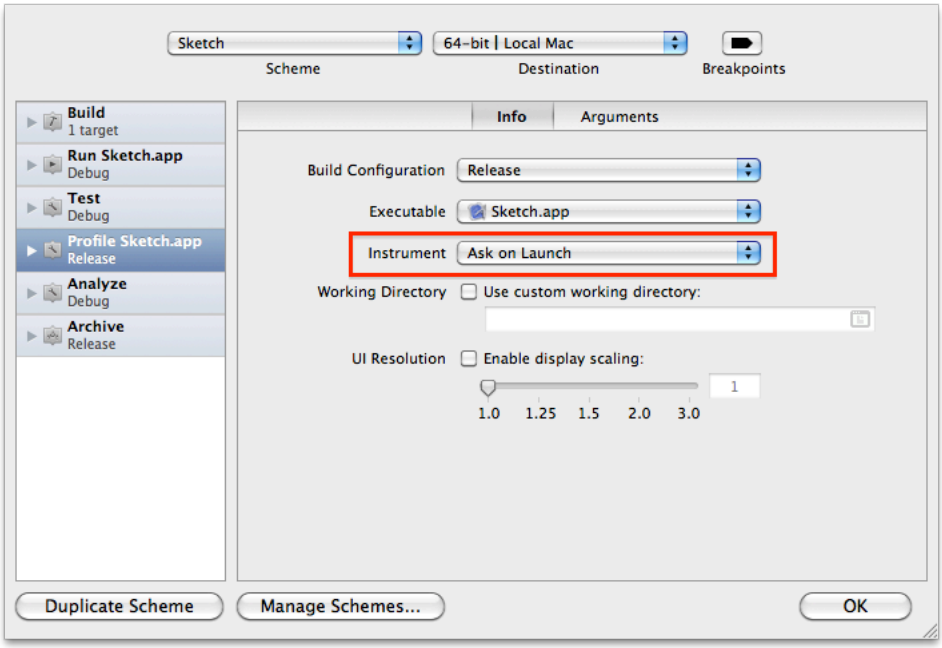


Running Instruments from your code allows you to gather information about your app every time you build it. Incorporating Instruments from the beginning of the app development process can save you time later by finding issues early in the development cycle.

To run Instruments while building your code

1. Open the scheme editor.
2. Select Edit Scheme.

3. Select Profile *app name*.



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

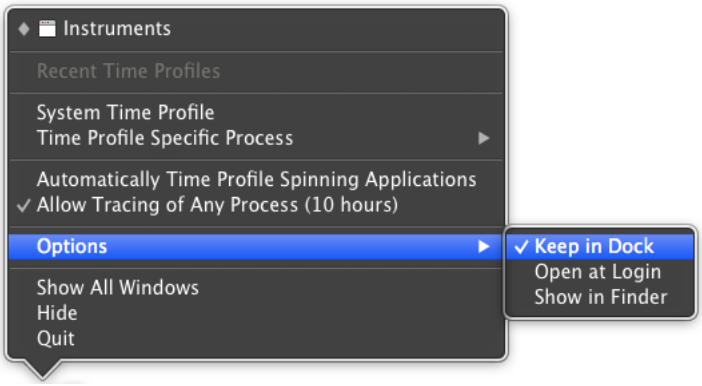
   The default choice is “Ask on Launch,” which causes Instruments to display its template chooser dialog when it starts up.

The third way to run Instruments can be achieved only after you have launched Instruments from Xcode. Save Instruments to the Dock, so that you are able to quickly run Instruments in the future without opening Xcode.

To run Instruments from the Dock

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

3. Click Options > Keep in Dock.



Gathering Your First Data

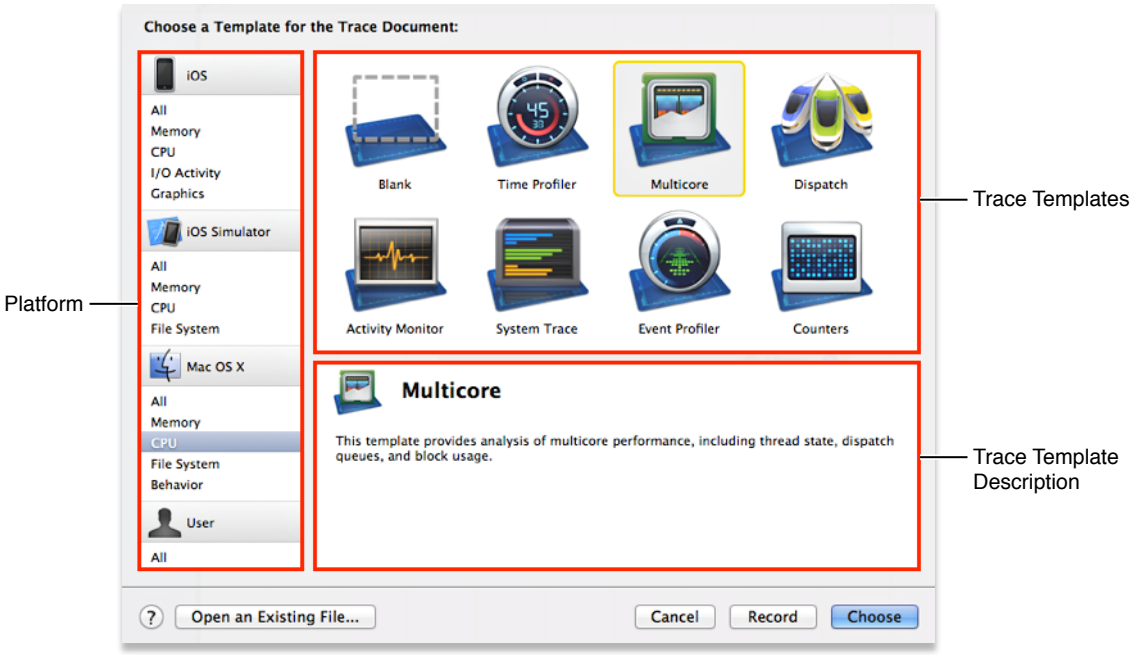
Even though each instrument is different, there is one general workflow when collecting information from your app. This workflow is a four-step process. You:

1. Choose a trace template
2. Direct Instruments to your app
3. Collect information from about your app
4. 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 individual instruments that are used together to provide you with a specific set of information. For Figure 1-1, the Multicore trace template is selected.

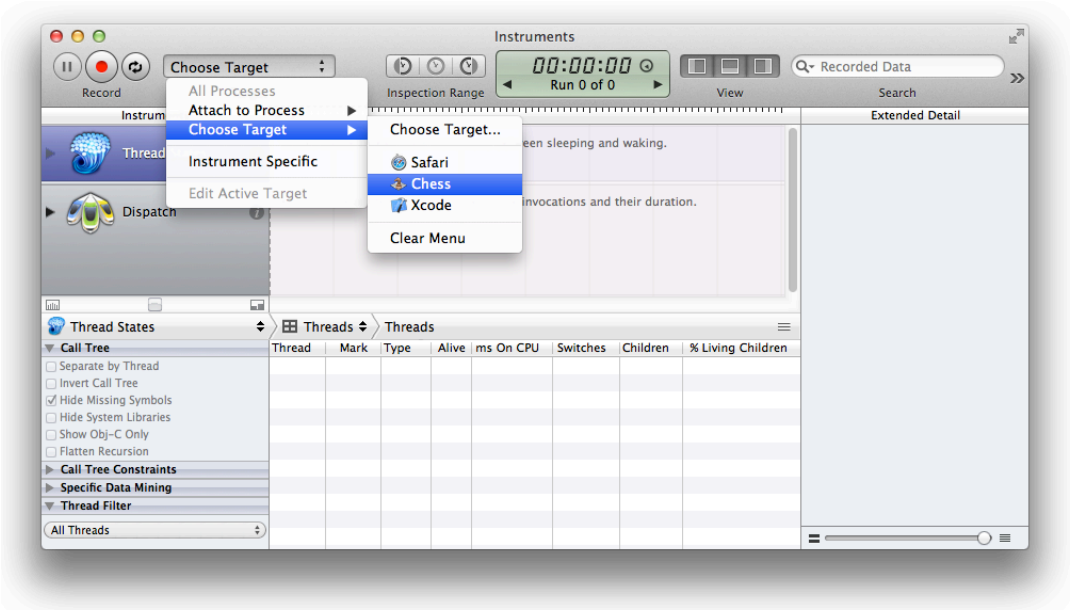
Figure 1-1 The Instruments Trace Template selection window



Choose Your Target

If you did not start Instruments while compiling your code, you must select a target for the instruments chosen. A target can be an app or a process that is already running. Some instruments allow you to gather information from all of the running processes on your device. In Figure 1-2, the Chess app is chosen.

Figure 1-2 Selecting the Chess app as the target

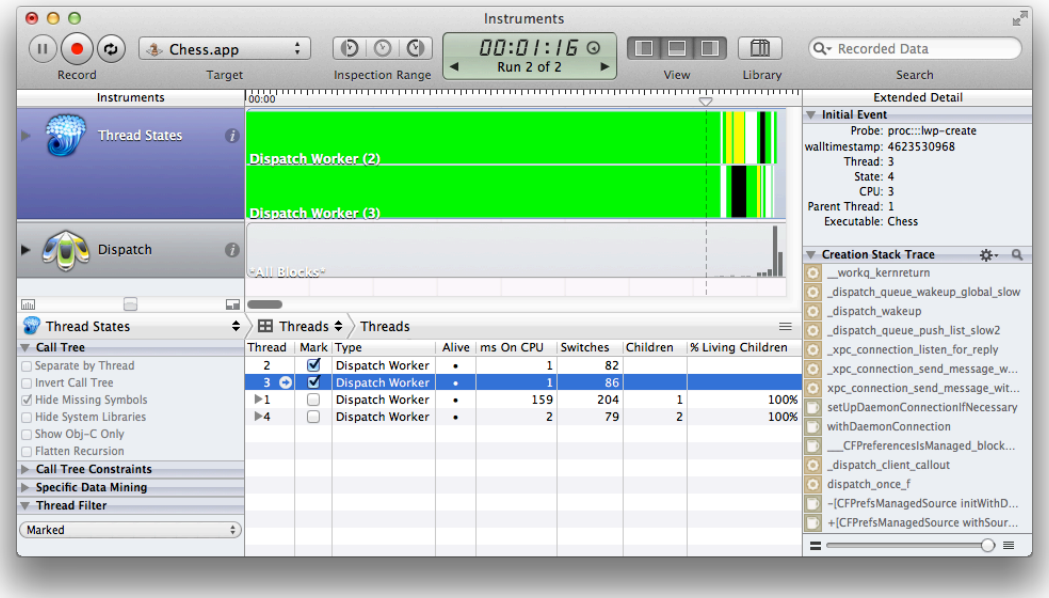


Collect and Examine the Data

Now that you have chosen the trace template and target for your app, it is time to collect the data. You can adjust the information that is collected by an instrument through its configuration dialog. Figure 1-3 (page 16) shows data collection using the default values for the two instruments. Click Record to start collecting data. When you have enough data, click Stop.

After collecting your data, you are able to examine it through the Detail pane. Selecting a row in the Detail pane brings up extra information on the row in the Extended Detail pane.

Figure 1-3 Collecting information with the Multicore trace template



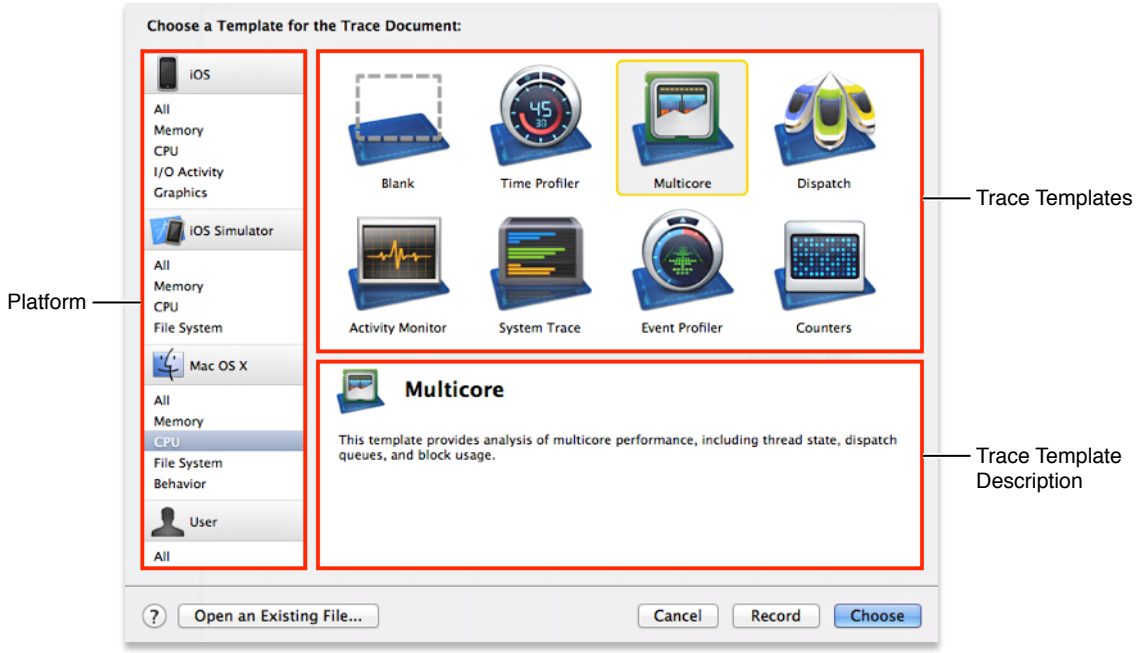
Touring Instruments

Instruments provides you with numerous ways to collect and examine information about your app. Instruments consists of two work areas, the trace template selection window and the trace document window. The trace template selection window, which appears when Instruments is first started, allows you to select a group of instruments based on their function. After you select a trace template, the trace document window appears. From this window, you can add more instruments, collect data, examine collected data, and much more. This chapter describes these sections 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 trace templates and platforms that are supported by Instruments on your machine. If you do not have the iOS SDKs installed, then you will not see the iOS platform displayed in the left column.

Figure 2-1 The Instruments window on opening



The opening screen is divided into three sections: platform, trace template, and trace template descriptions. Table Table 2-1 (page 18) describes what is found in each of these sections.

Table 2-1 Opening screen sections

Feature	Description
Platform	There are three possible platforms available for Instruments: iOS, iOS Simulator, and OS X. Each platform contains its own set of trace templates that are configured to run on that platform. Certain instruments can run on any platform, but many are platform specific. You will see only those platforms you have downloaded the SDK for.
Trace template	Trace templates are groups of instruments that have been collected together in order to achieve a specific task. The trace templates shown change as you select the platform and group type.
Trace template description	The trace template description section provides a brief overview of what the type information that can be collected by the instruments contained within the trace template.

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. 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 window while tracing

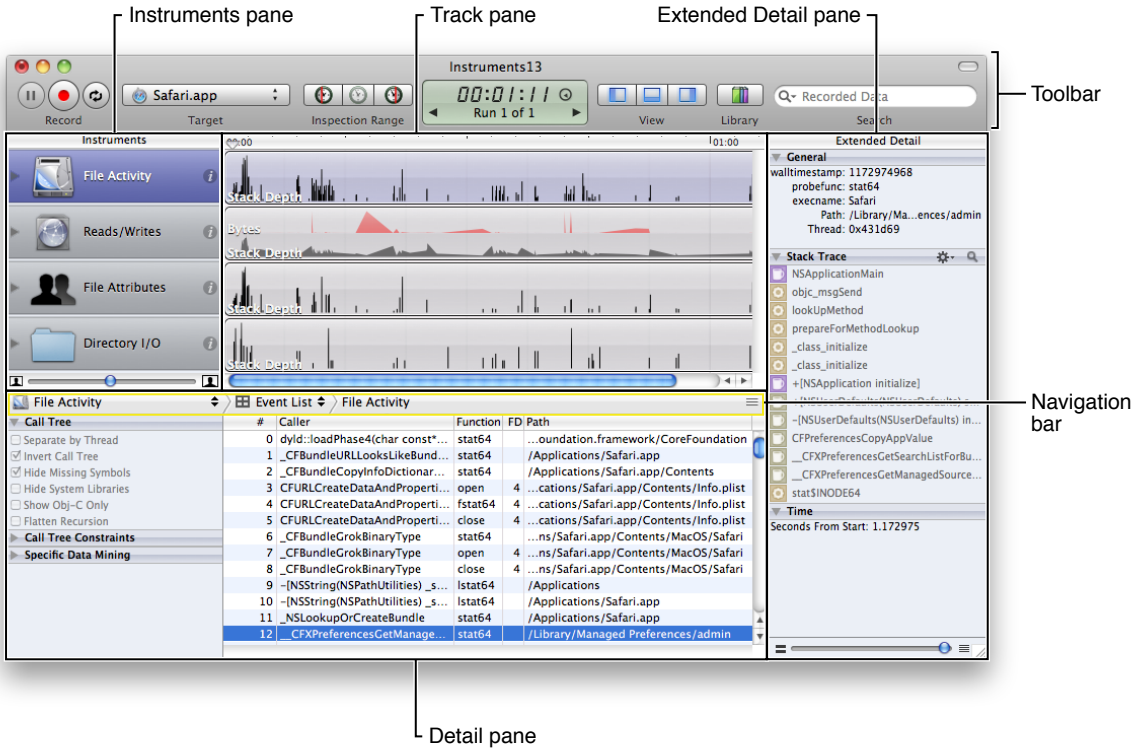


Table 2-2 lists some of the key features that are called out in Figure 2-2 (page 19) and provides a more in-depth discussion of how you use that feature.

Table 2-2 Trace document key features

Feature	Description
Instruments pane	This pane holds the instruments you want to run. You can drag instruments into this pane or delete them. You can click the inspector button in an instrument to configure its data display and gathering parameters.
Track pane	The track pane displays a graphical summary of the data returned by the current instruments. Each instrument has its own "track," which provides a chart of the data collected by that instrument. Although this pane's information is read-only, it is in this pane that you select data points for closer examination.

Feature	Description
Detail pane	The Detail pane shows the details of the data collected by each instrument. Typically, this pane displays the explicit set of "events" that were gathered and used to create the graphical view in the track pane. If the current instrument allows you to customize the way detailed data is displayed, those options are also listed in this pane.
Extended Detail pane	The Extended Detail pane shows even more detailed information about the item currently selected in the Detail pane. Most commonly, this pane displays the complete stack trace, timestamp, and other instrument-specific data gathered for the given event.
Navigation bar	The navigation bar shows you where you are and how you got there. It includes two menus, the active instrument menu and the detail view menu. You can click entries in the navigation bar to select the active instrument and the level and type of information in the detail view.

The trace document's toolbar lets you add and control instruments, open views, and configure the track pane. Figure 2-3 identifies the different controls on the toolbar, and Table 2-3 provides detailed information about how you use those controls.

Figure 2-3 The Instruments toolbar

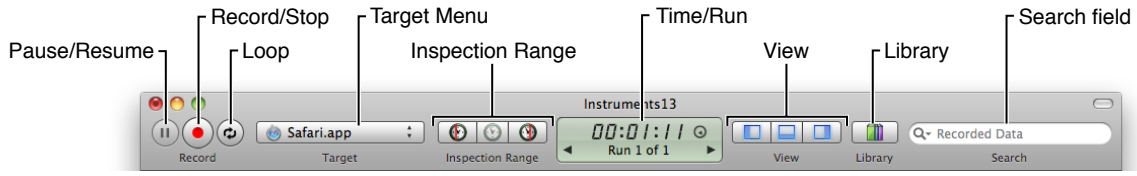


Table 2-3 Trace document toolbar controls

Control	Description
Pause/Resume button	Pauses the gathering of data during a recording. This button does not actually stop recording, it simply stops Instruments from gathering data while a recording is under way. In the track pane, pauses show up as a gap in the trace data.
Record/Stop button	Starts and stops the recording process. Use this button to begin gathering trace data.
Loop button	Sets whether the user interface recorder should loop during play back to repeat the recorded steps continuously. Use this setting to gather multiple runs of a given set of steps.

Control	Description
Target menu	Selects the target for the document. The target is the process (or processes) for which data is gathered.
Inspection Range control	Selects a time range in the track pane. When set, Instruments displays only data collected within the specified time period. Use the buttons of this control to set the start and end points of the inspection range and to clear the inspection range.
Time/Run control	Shows the elapsed time of the current trace. If your trace document has multiple data runs associated with it, you can use the arrow controls to choose the run whose data you want to display in the track pane.
View control	Hides or shows the Instruments pane, Detail pane, and Extended Detail pane. This control makes it easier to focus on the area of interest.
Library button	Hides or shows the instrument library. For information on using the Library window, see "Adding and Configuring Instruments" (page 22).
Search field	Filters information in the Detail pane based on a search term that you provide. Use the search field's menu to select search options.

Adding and Configuring Instruments

The Instruments tool uses 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 document, most documents contain fewer than ten for performance reasons. You can even include the same instrument multiple times, with each instrument version configured to gather data from a different process on the system.

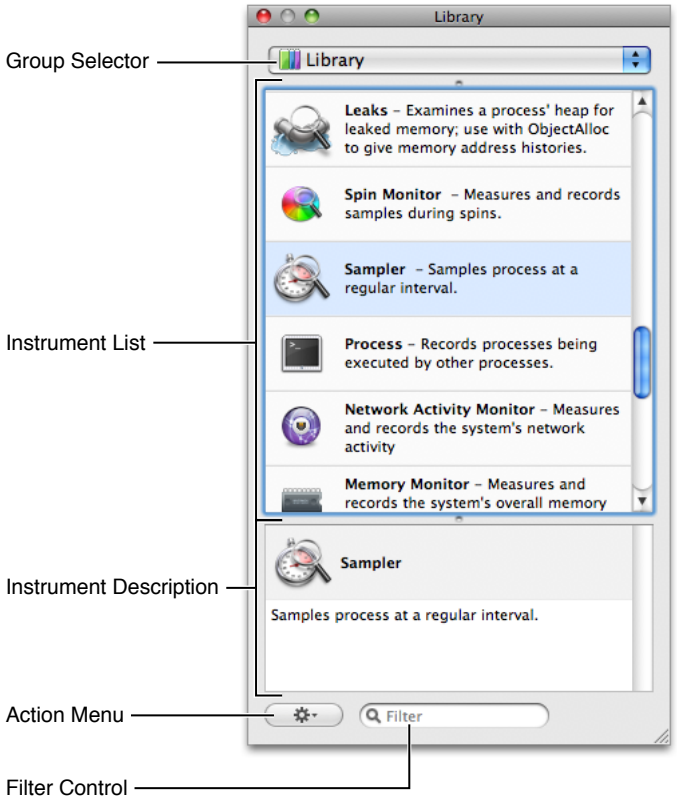
Instruments comes with a wide range of built-in instruments whose job is to gather specific data from one or more processes. Most of these instruments require little or no configuration to use. You simply add them to your trace document and begin gathering trace data. You can also create custom instruments, 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 101).

Viewing the Built-in Instruments with the Library Window

The instrument library (shown in Figure 3-1) displays all of the instruments that you can add to your trace document. The library includes all of the built-in instruments plus any custom instruments you have defined.

Figure 3-1 The instrument library



To open the Library window

Do one of the following:

- Click the Library button in the toolbar.
- Choose Window > Library.

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 detail pane.

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.

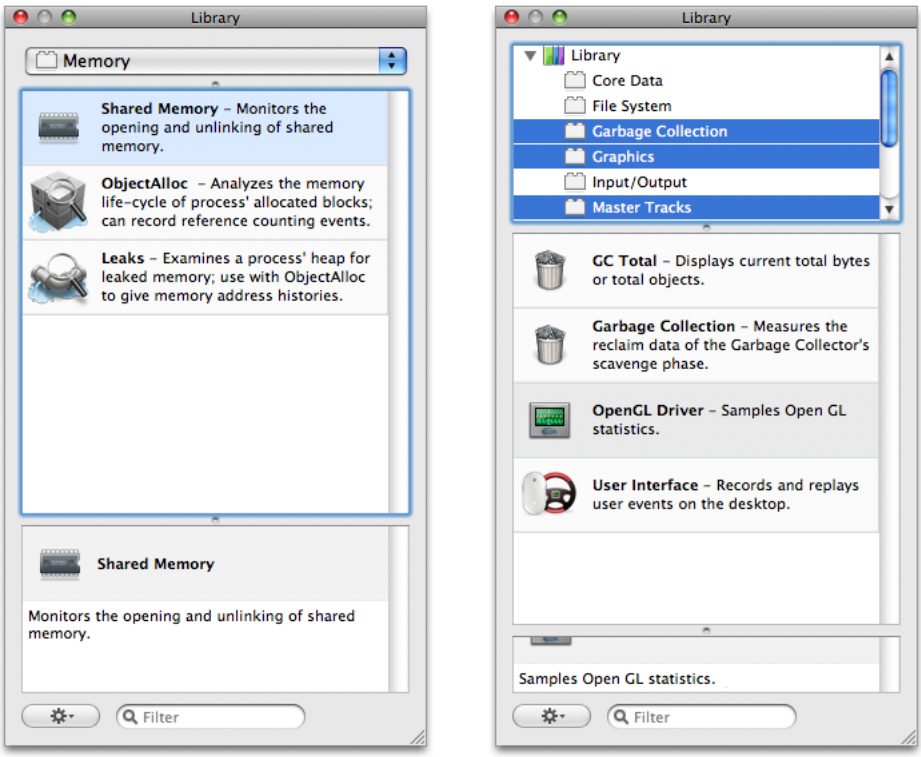
Finding an Instrument in the Library

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.

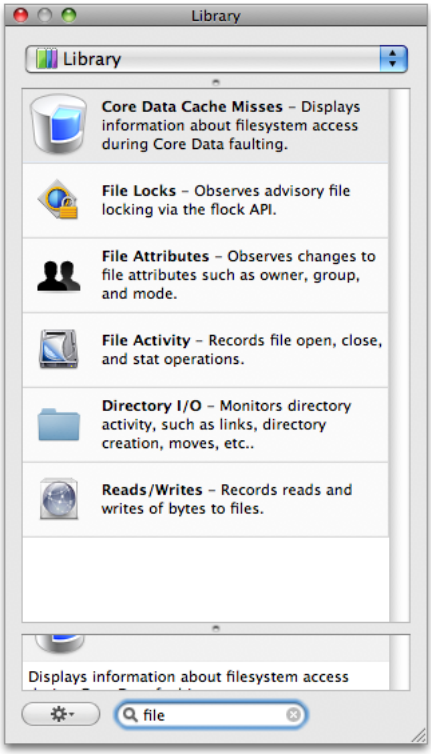
Figure 3-2 shows both standard mode and outline mode in the Library window. The window on the left shows standard mode, in which you select a single group using the pop-up menu. The window on the right shows the outline view, in which you can select multiple groups and manage your own custom groups.

Figure 3-2 Viewing an instrument group



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

Figure 3-3 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.

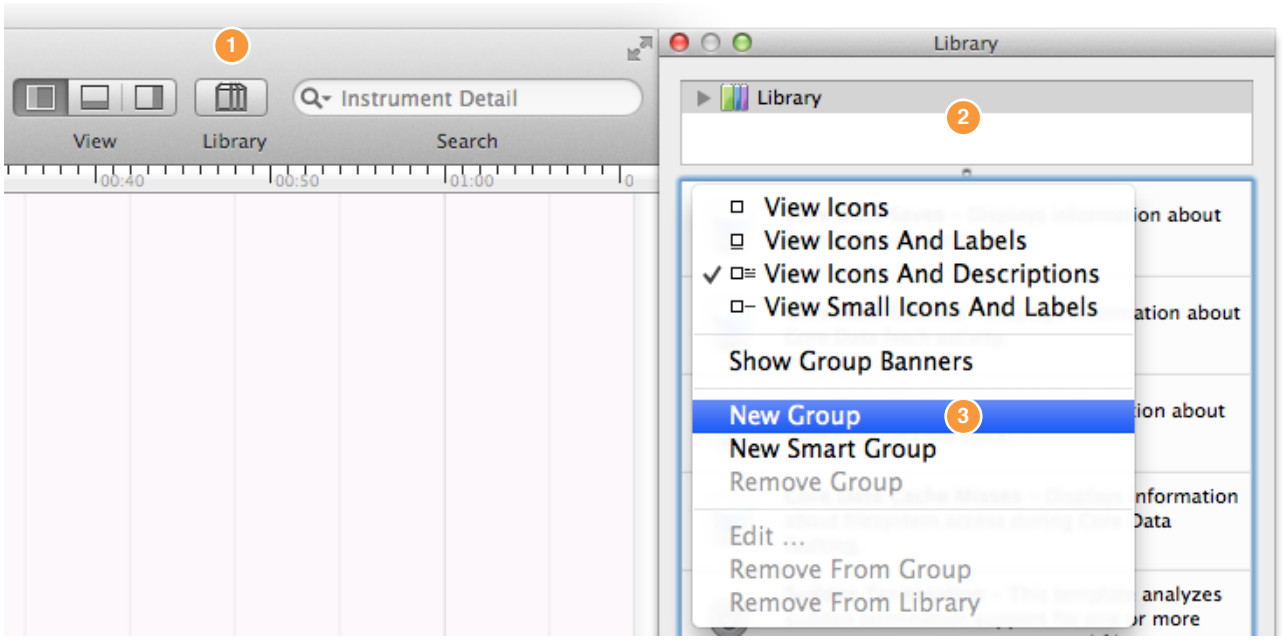
Creating a Custom Group

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

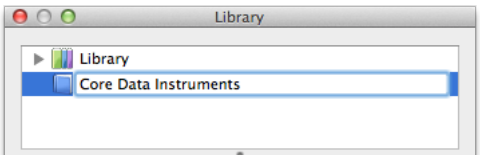
To create a static group in the Library window

1. Click Library to open the Library window.
2. Control-click in the Library window.

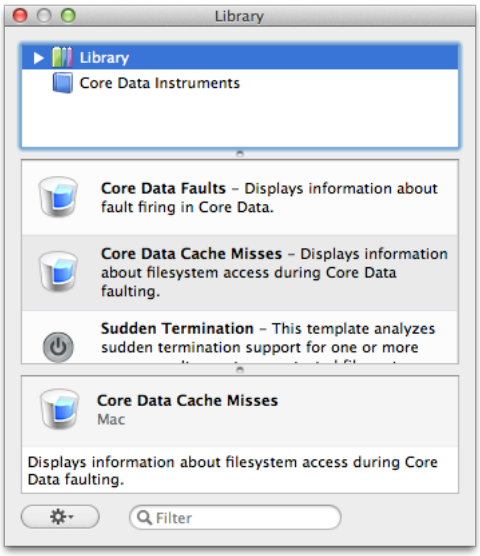
3. Choose New Group.



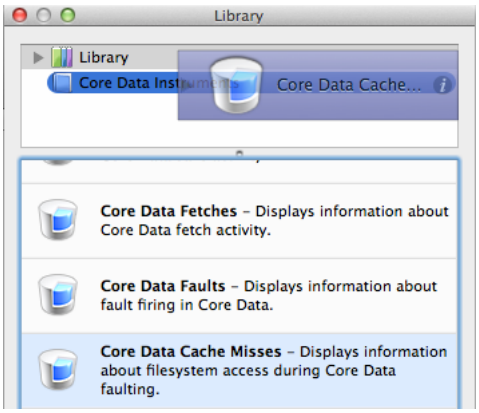
4. Enter a name for the group and press Return.



5. Select the Library directory.



6. Drag the desired instruments into the new group.



When you first open the Library window, the library directory and all previously created groups are displayed. The library directory contains a complete list of all available instruments.

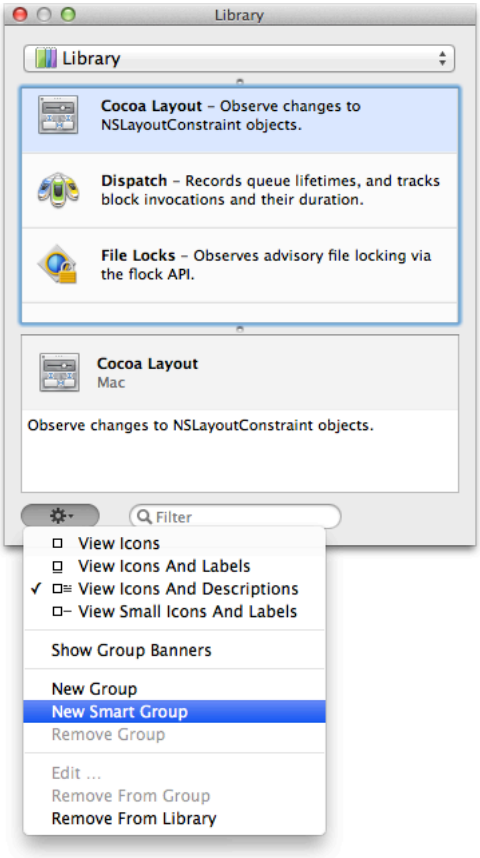
Populate a new group with any number of instruments from the library directory. Give the group a name that identifies its purpose so you can easily select it when needed. You can nest groups in a hierarchical structure. You can also drag a nested group outside the structure to stand alone. If an instrument 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.

Creating a Smart Group

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

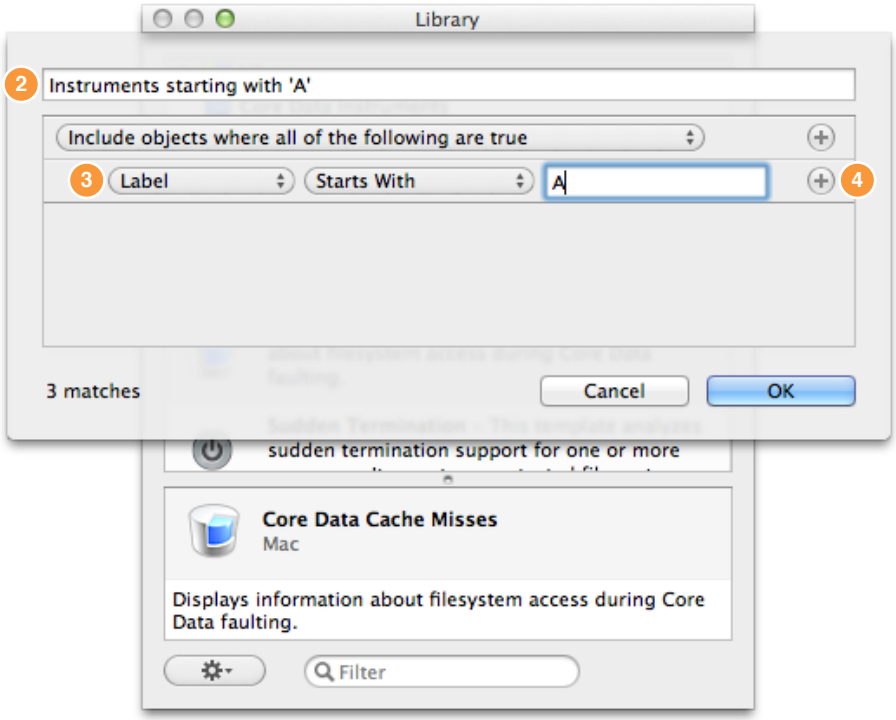
To create a smart group

1. Choose New Smart Group from the Library window's action menu.

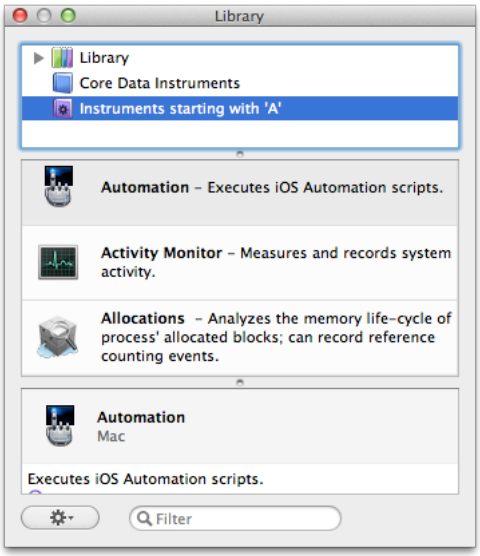


2. Enter a name for the group in the Label field.
3. Specify a rule for the group.

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



5. Verify that the 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. This criterion supports the comparison operators Starts With, Ends With, and Contains.
Used Within	Matches instruments based on when they were used. You can use this criterion to match only 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 the specified string.
Category	Matches instruments whose library group name matches the specified string. This criterion 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. Instruments displays the rule editor again so that 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. If you are currently viewing groups using the outline view, you can also select the group and press the Delete key.

Adding an Instrument

Expand on the amount and type of data collected by adding a new instrument to your trace document.

To add an instrument from the Library window

1. Click Library to open the Library window.
2. Select the Library directory.
3. Double-click an instrument to add it to the trace document.

You can place multiple instances of an instrument in your trace document with each one set up to collect data from a different process or app. In this way, you can track how a client and server interact with each other.

Configuring an Instrument

Most instruments are ready to use as soon as you add them to the Instruments pane. Some instruments can also be configured using the instrument inspector. The content of the inspector varies 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

Do one of the following:

- Click the inspector icon to the right of the instrument's name.
- Choose File > Get Info.
- Press Command-I.

Controls related to displaying information in the track pane can be configured before, during, or after your record data for the track. Instruments automatically gathers the data it needs for each display option regardless of whether that option is currently displayed in the track pane.

For a list of instruments, including their configurable options, see *Instruments User Reference*.

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.

Setting Up Data Collection with the Target Pop-up Menu

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

The Target pop-up menu provides you with three choices for collecting data:

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

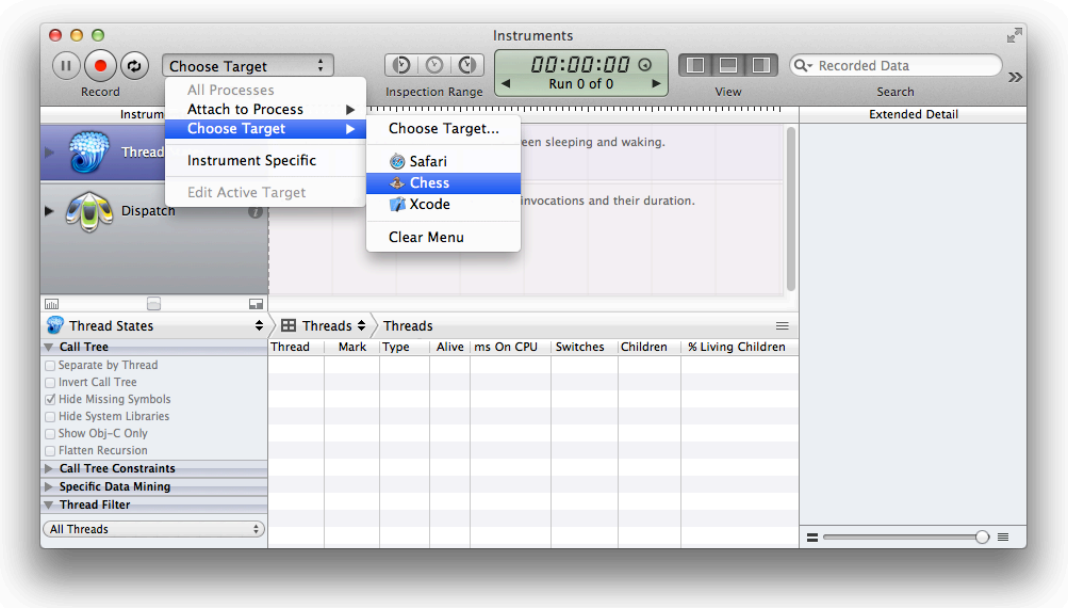
Typically, users collect data from one app at a time. When you target a single app, all of the instruments in the trace document collect data from the targeted app.

To target a single app

1. Click Choose Target from the Target pop-up menu.
2. Click Choose Target to search for an app

   Previously targeted apps are shown below the Choose Target option.

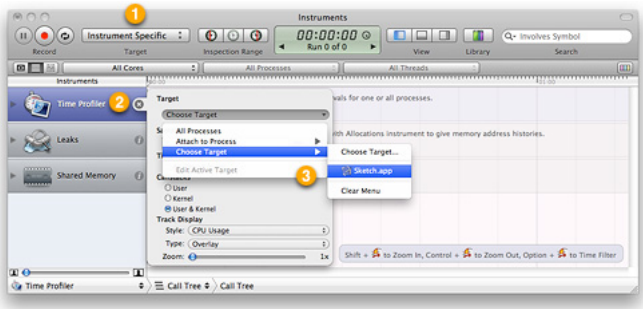
3. Select your app and press the Choose button.



You can set each instrument in a trace document to collect data from a particular app, process, or device, independent of the other instruments. In this way, you can collect data on how multiple apps interact with each other.

To set different targets for multiple instruments

1. Choose Instrument Specific from the Target pop-up menu.
2. Click the instrument's inspector icon.
3. Choose the targeted app or process from the pop-up menu in the Target section.



You can add multiple versions of the same instrument to your trace document by dragging another copy of the instrument from the Library into the document to record data from a different app, process, or device with each version of the instrument.

You can collect various streams of data simultaneously. For example, you could simultaneously collect data from:

- All processes with one instrument.
- A single app with a second instrument
- A different app with a second version of the second instrument
- A third app with a third instrument

For iOS development, the pop-up menu for the target selection includes only the targets you can profile. For OS X development, the pop-up menu includes the targets you have already profiled.

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

To enable a wireless device

1. Make sure your mobile device is connected by a USB cable.
2. Press the Option key and click the Target pop-up menu.
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.

See the HTML version of this document to view the video.

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: Bonjour and multicast must be enabled on your wireless network access point.

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

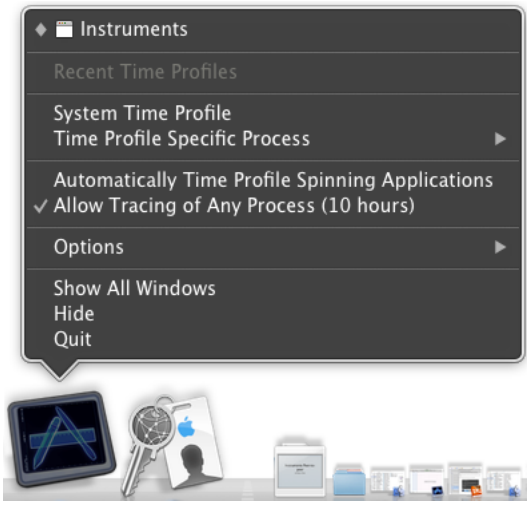
Save time by running Time Profiler from the Instruments app icon in the Dock to record events while Instruments is not running. Record fleeting or transient events. You can profile the following:

- **System Time Profile.** Starts profiling all system processes.
- **Time Profile Specific Process.** Starts the Time Profiler instrument with the targeted 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. Control-click the Instruments dock icon.
2. Choose the call stack to profile.
3. Click All Thread States to profile all available threads.

4. 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:

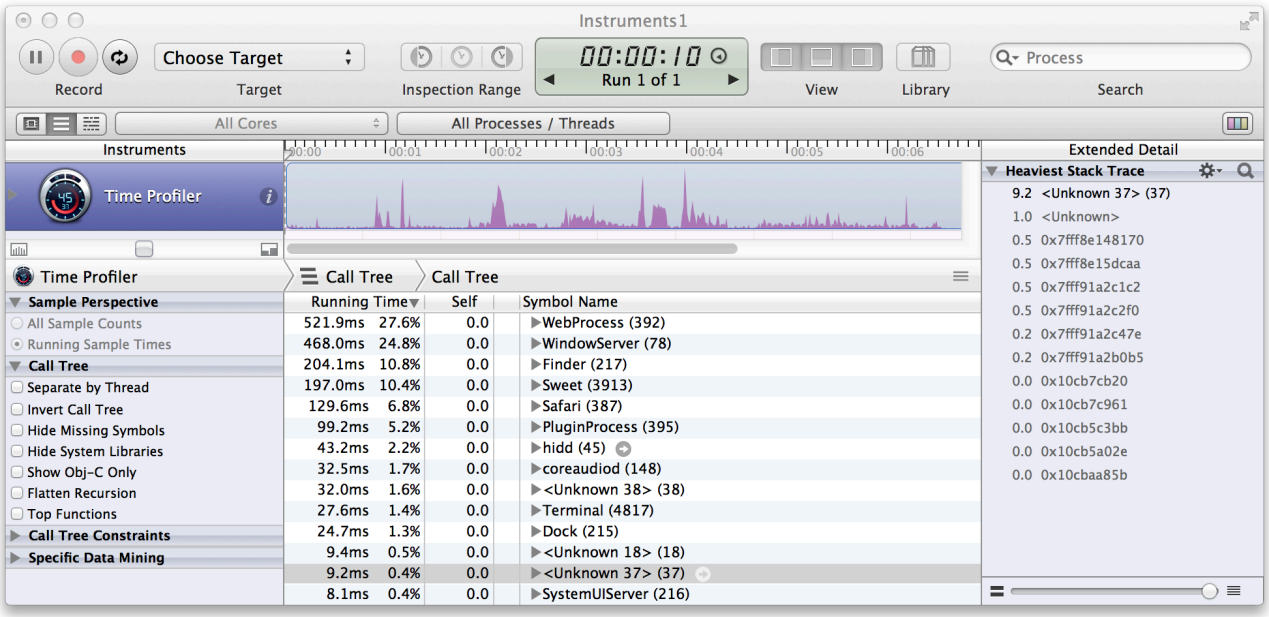
- **Activity Monitor.** Monitors overall system activity and statistics, including cpu, memory, disk, and network. Activity Monitor also monitors all existing processes and parent/child process hierarchies.
- **Allocations.** Measures heap memory usage by tracking allocations, including specific object 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 via 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 trace templates and populates them with the collected data. You can view and manipulate the data to locate any issues with your app.

Figure 4-1 Imported Time Profiler data



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

Command	Description
`-l`	Provides a list of all supported templates.
`-L`	Provides a list of all supported templates and a description of what each template does.
`-legacy`	Executes the legacy Instruments command-line interface found in `/usr/bin/instruments`.

Command	Description
`–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 template 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.
`–userandkernelstacks`	Backtraces include both kernel and user stacks. If neither `–kernelstacks` or `–userandkernelstacks` options are specified, backtraces include user stacks only.
`–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.

Command	Description
`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 templates. 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 template. 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 templates. 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 templates. 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 templates. 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 clicking the inspector icon for that instrument.

Running Instruments in Deferred Mode

Increase the accuracy of performance-related data by deferring data analysis until you quit the application you are testing. Instruments analyzes and displays data while your app runs, allowing you to view the data as it is collected. Running Instruments in deferred mode delays the analysis of data until the data collection is done, either after your application has finished running or after you click Stop. While in deferred mode, you are blocked from interacting with the instruments that are collecting data.

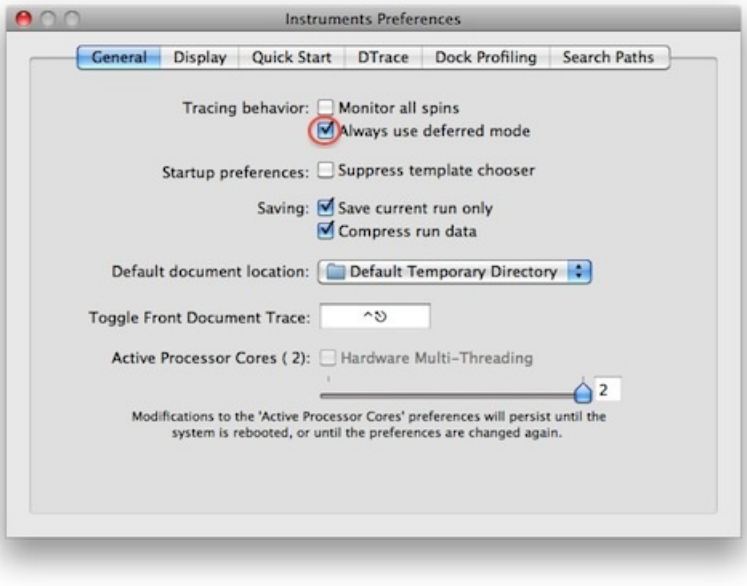
In deferred mode, after Instruments has finished collecting data, Instruments processes the data and displays it onscreen. Even though deferring the data analysis adds time to the back end of the data collection process, it helps ensure that performance-related data is accurate.

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.

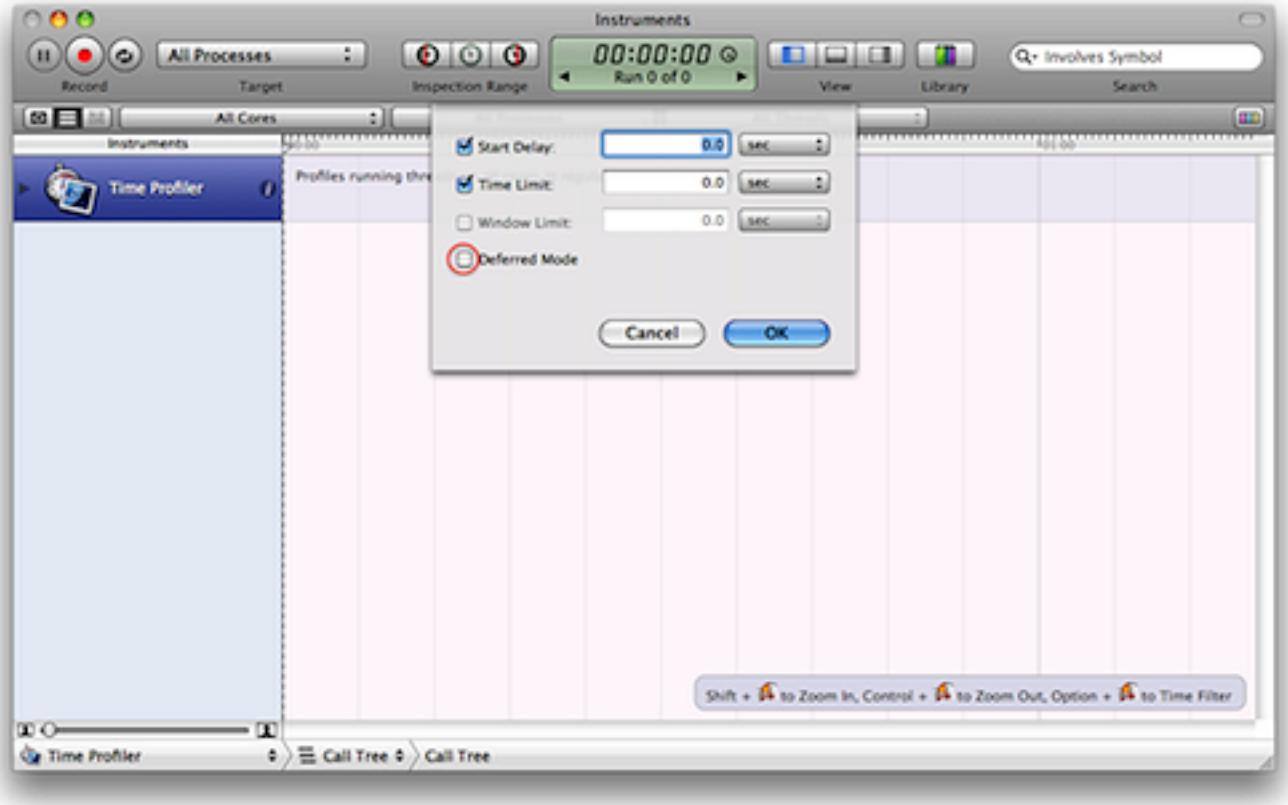


Using deferred mode moves the data processing to after the collection of data, resulting in a delay at the end of data collection as Instruments processes the data. In the event of especially long traces, this delay can be significant. To avoid this delay, you can set deferred mode for only those traces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

Resymbolicating Your Data

Instruments requires accurate information about your project in order to provide you with the best results. This is accomplished by ensuring that the system is able to see all of the symbols associated with your document. Resymbolicate your document when addresses, rather than symbols, are displayed in trace documents. Symbolication maps addresses to their associated symbols and line number information.

Resymbolication is often necessary when building on one machine and testing performance on another. 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.

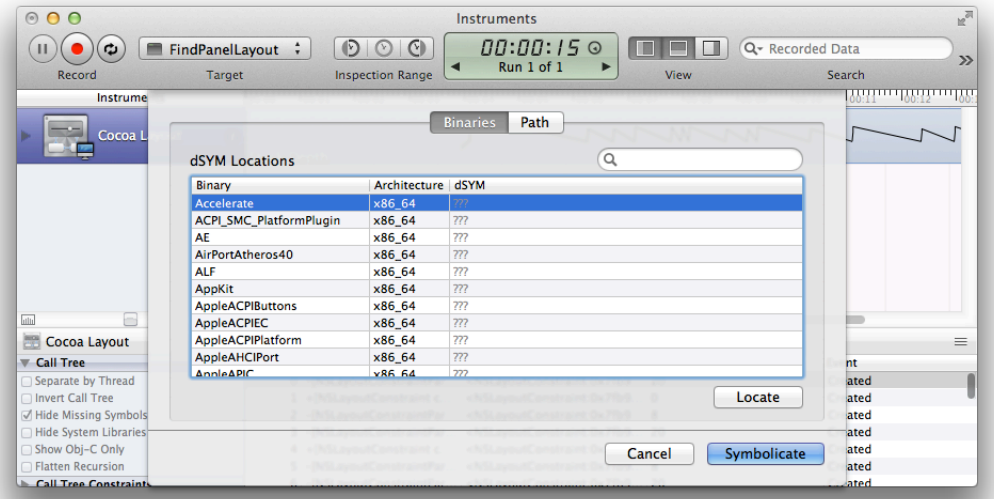
To resymbolicate your app

1. Choose File > Re-Symbolicate.

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

2. In the Re-Symbolicate dialog that appears, click the Binaries tab.
3. Highlight the binary (executable) from which your trace document was made.
4. Press the Locate button.

5. Click Symbolicate.



Viewing the Collected Data in the Track Pane

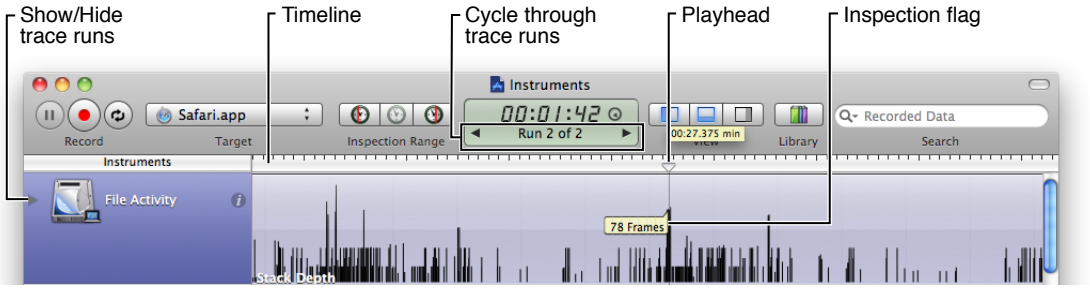
The most prominent portion of a trace document window is the track pane. The track pane occupies the area immediately to the right of the instruments pane. This pane presents a high-level graphical view of the data gathered by each instrument. You use this pane to examine the data from each instrument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

The graphical nature of the track pane makes it easier 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 Spin Monitor instrument can also point out places where your app becomes unresponsive. If the graph for Spin Monitor is relatively empty, you know that your app is being responsive, but if the graph is not empty, you might want to examine why that is.

Figure 5-1 shows a sample trace document and calls out the basic features of the track pane. You 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 48).

Figure 5-1 The track pane



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 features give you the option to display the 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 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 refine the detail presented.

To zoom in and out of your data

Do one of the following:

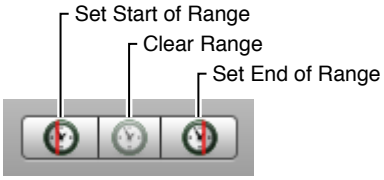
- Press the Shift key, and drag across a section of data to zoom in.

- Press the Control key, and drag across a section of data to zoom out.

Viewing Data for a Range of Time

Although zooming in on a particular event in the track pane lets you see what happened at a specific time, you may also be interested in seeing the data collected over a range of time. You use the Inspection Range control (shown in Figure 5-2) to focus on the data collected in a specific time range.

Figure 5-2 Inspection Range control



To mark a time range for inspection

1. Set the start of the range.
   a. Drag the playhead to the desired starting point in the track pane.
   b. Click the leftmost button in the Inspection Range control.
2. Set the end of the range.
   a. Drag the playhead to the desired endpoint in the track pane.
   b. Click the rightmost button in the Inspection Range control.

Instruments highlights the contents of the track pane that fall within the range that 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.

You can also set an inspection range by holding the Option key and dragging in the track pane of the desired instrument. 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.

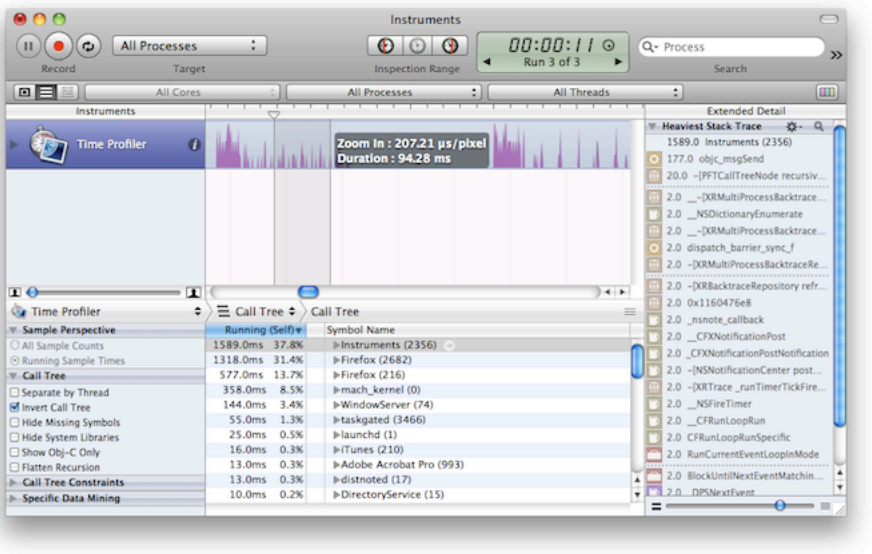
When you set a time range, Instruments filters the contents of the Detail pane, showing data collected only within the specified range. You can quickly narrow down the large amount of information collected by Instruments and see only the events that occurred over a certain period of time.

To clear an inspection range, click the Clear Range button in the center of the Inspection Range control.

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 5-3 shows a section of the track pane highlighted before it is zoomed in on.

Figure 5-3 Zooming in on a section of data

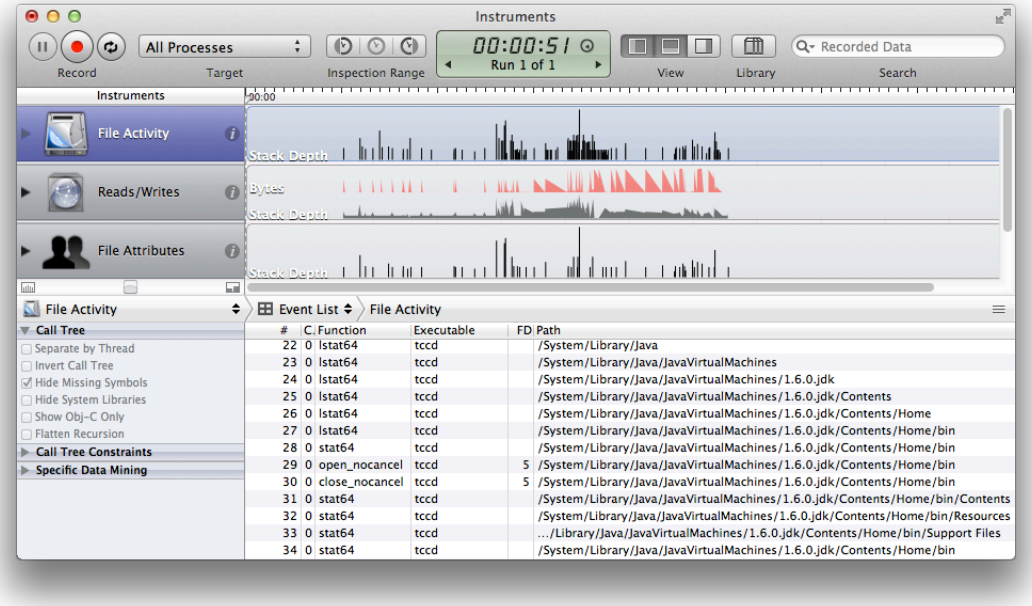


Examining Data in the Detail Pane

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 5-4 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. For information about what each instrument displays in the Detail pane, see *Instruments User Reference*.

Figure 5-4 The Detail pane



To open or close the Detail pane

Do one of the following:

- Choose View > Detail.
- Click the middle View button in 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. See *Instruments User Reference* for information on each individual instrument's Detail pane display options.

To view an instrument's data using one of these formats, choose the appropriate mode from the rightmost menu in the navigation bar. Which modes an instrument supports depends on the type of data gathered by that instrument.

You can use disclosure triangles in the appropriate 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

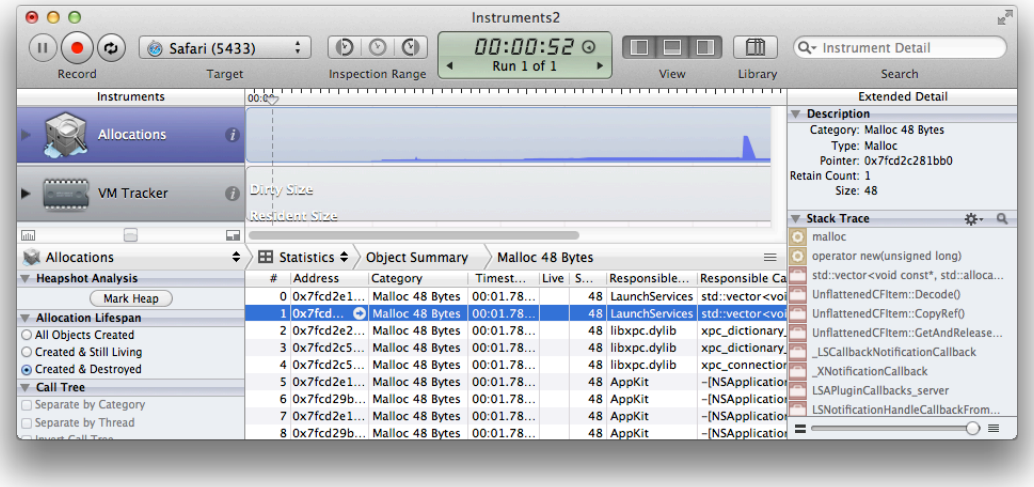
You can sort the information displayed in the Detail pane according to the data in a particular column. To do so, click the appropriate column header. The columns in the Detail pane differ with each instrument.

Working in the Extended Detail Pane

For some instruments, the Extended Detail pane shows additional information about the item currently selected in the Detail pane. The Extended Detail pane 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 5-5 shows the Extended Detail pane 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 5-5 Extended Detail pane



To open or close the Extended Detail pane

Do one of the following:

- Choose View > Extended Detail.
- Click the Extended Detail View button in the toolbar.

You can configure the information shown in the stack trace using the Action menu at the top of that section. Clicking and holding the Action menu icon displays a menu from which you can enable or disable the options in Table 5-1.

Table 5-1 Action menu options

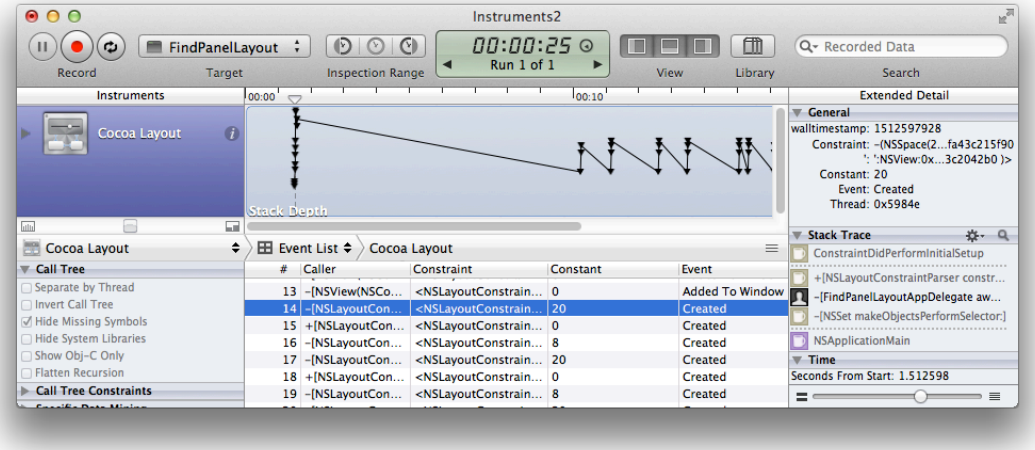
Action	Description
Invert Stack	Toggles the order in which calls are listed in the stack trace.
Source Location	Displays the source file that defines each symbol whose source you own.
Library Name	Displays the name of the library containing each symbol.
Frame #	Displays the number associated with each frame in the stack trace.
File Icon	Displays an icon representing the file in which each symbol is defined.
Trace Call Duration	Creates a new Instruments instrument that traces the selected symbol and places that instrument in the Instruments pane.
Look up API Documentation	Opens the Xcode Documentation window and brings up documentation, if available, for the selected symbol.
Copy Selected Frames	Copies the stack trace information for the selected frames to the pasteboard so that you can paste it into other apps.

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.

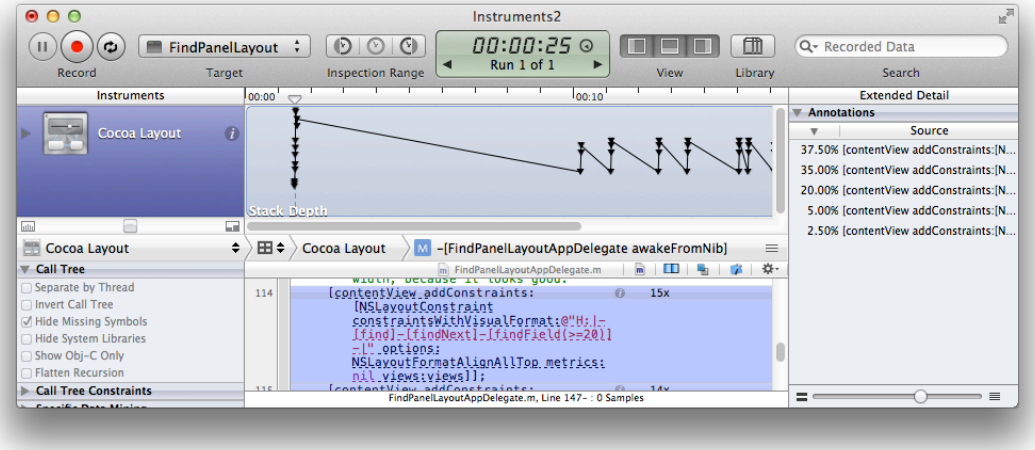
To see your personal code

1. Click on a line in the Detail pane.

2. Double-click the symbol name in the Extended Detail pane.



3. Click the Xcode icon in the top right of the Detail pane to make changes.



You can collapse stack traces by using the slider control at the bottom of the Extended Detail pane. This control governs backtrace compression with filtering. Its purpose is to reduce the detail in the stack trace and to display only what matters.

Saving and Importing Trace Data

Instruments provides several ways for you to save instrument and trace data. For a given Instruments document window, you can save the trace data you've recorded with that document or you can save the instrument configuration of that document. Saving the trace data lets you maintain a record of your app's performance over time. Saving the document configuration avoids the need to recreate a commonly used configuration each time you run Instruments.

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

Saving a Trace Document

At times you will 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. Click Save.

By default, Instruments saves only the trace data collected on the most recent run. If you have recorded multiple runs and want Instruments to save all of that data, you must deselect the Save Current Run Only option in the General pane of Instruments preferences before saving the document.

Saving an Instruments Trace Template

During your development cycle, you may want to gather data at several points by running Instruments on your app using a fixed set of instruments. Rather than reconfigure the same set of instruments in your trace document each time you run Instruments, you can configure the trace document once and save it as a trace template.

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.
4. Select an icon for the template.
5. Enter a description for the template.
6. Click Save.

Trace template documents are not the same as the Instruments templates that appear when you create a new document. You open a trace template in the same way you open other Instruments documents, by choosing File > Open. When you open a trace template, Instruments creates a new trace document with the template configuration but without any data.

Xcode supports launching your apps using your custom trace templates. To add your trace template to the Run menu in Xcode, drop the template in the `/Users/<username>/Library/Application Support/Instruments/Templates` directory on your local system.

Exporting Track Data

Instruments lets you export trace data to a comma-separated value (CSV) file format. This simple data file format is supported by many apps. For example, you might save your trace data in this format so that you can import it into a spreadsheet app.

To save your trace data to a CSV file, select the instrument whose data you want to export and choose Instrument > Export Data for: <*Instrument Name*>. Instruments exports the data for the most recent run of that 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. And although you can use the Call Tree configuration options of the Sampler instrument to trim the sample data, you cannot prune data using the Call Tree Constraints or Inspection Range controls. 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. Click Save.

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 114).

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. This chapter describes how to identify common memory issues, from memory leaks to zombies.

Examining Memory Usage with the Activity Monitor Trace Template

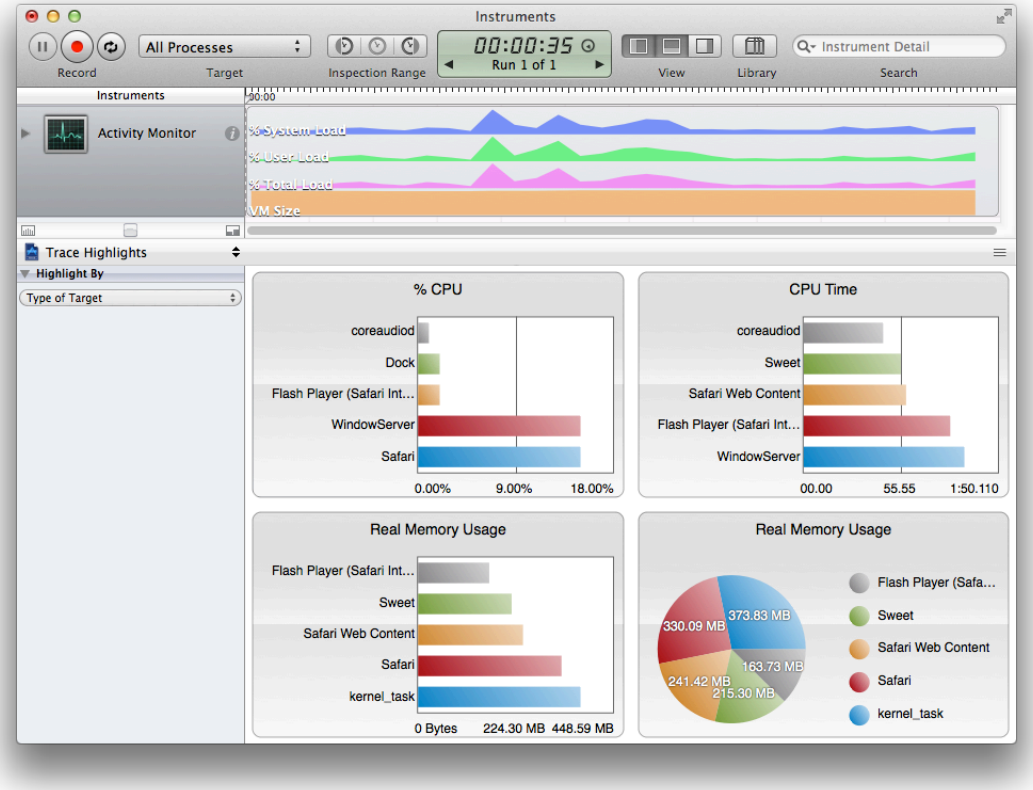
The Activity Monitor trace template monitors overall system activity and statistics, including CPU, memory, disk, and network. It also monitors all existing processes and can be used to attach new instruments to specific processes, monitor parent-child process hierarchies, and to quit running processes. It consists of the Activity Monitor instrument only. You'll see later that the Activity Monitor is also used to monitor network activity on iOS devices.

The Activity Monitor instrument provides you with four convenient charts for a quick, visual representation of the collected information. The two charts that 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 following configuration options provide memory-specific information through the Activity Monitor. For statistic definitions and complete configuration options, see “Activity Monitor Instrument” in *Instruments User Reference*.

- 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

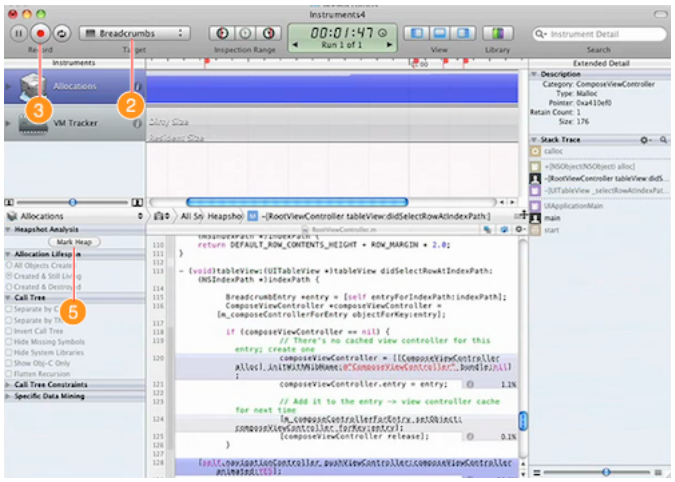
Recovering Memory You Have Abandoned

The Allocations trace template measures heap memory usage by tracking allocations, including specific object allocations by class. It also records virtual memory statistics by region. It consists of the Allocations and the VM Tracker instruments.

Avoid abandoned memory by ensuring that the heap does not continue to grow when the same set of operations are continuously repeated. For example, opening a window then immediately closing it, or setting a preference then immediately unsetting it are operations that conceptually return the app to a previous and stable memory state. Cycling through such operations many times should not result in unbounded heap growth. To ensure that none of your code abandons memory, repeat user scenarios and use the Mark Heap feature after each iteration. After the first few iterations (where caches may be warmed), the persistent memory of these iterations should fall to zero. If persistent memory is still accumulating, select the focus arrow to see a call tree of the memory. There you can identify the code paths responsible for abandoning the memory. Ensure that your scenarios exercise all your code that allocates memory.

To find memory abandoned by your app

1. Open the Allocations template.
2. Choose your app from the Choose Target pop-up menu.
3. Click the Record button.
4. Repetitively perform an action in your app that starts from, and finishes in, the same state.
5. After each iteration of the repeated action, click the Mark Heap button to take a snapshot of the heap.
6. Repeat steps 4 and 5 until you see whether the heap is growing without limit, and then click the Stop button.
7. Analyze objects captured by the heapshots to locate abandoned memory.



If the heap continues to grow after the first few iterations of the cycle, you know your app is abandoning memory. Find a heapshot that seems representative of the repeated heap growth. Click the focus button to the right of the heapshot name to display objects created during that time range that are still living after the app has executed.

After you stop the trace, you can still take snapshots by dragging the inspection head in the trace window timeline to where you want the snapshot, and clicking Mark Heap. After stopping the trace, take one last snapshot at the end of the trace. At that point, the number of persistent objects should be zero.

Note: Garbage collection does not release abandoned memory.

Finding Leaks in Your App

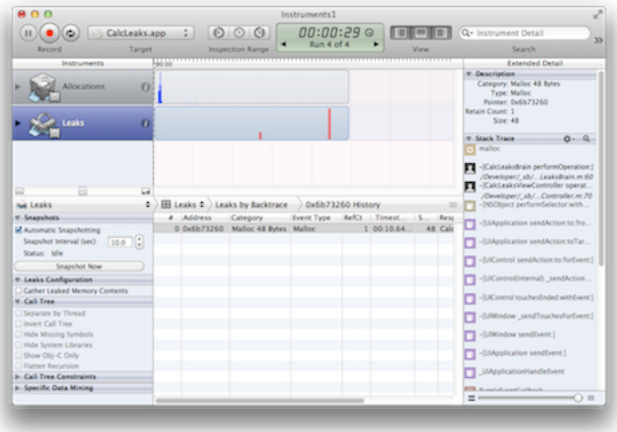
The Leaks trace template measures general memory usage, checks for leaked memory, and provides statistics on object allocations by class as well as memory address histories for all active allocations and leaked blocks. It consists of the Allocations and Leaks instruments.

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

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

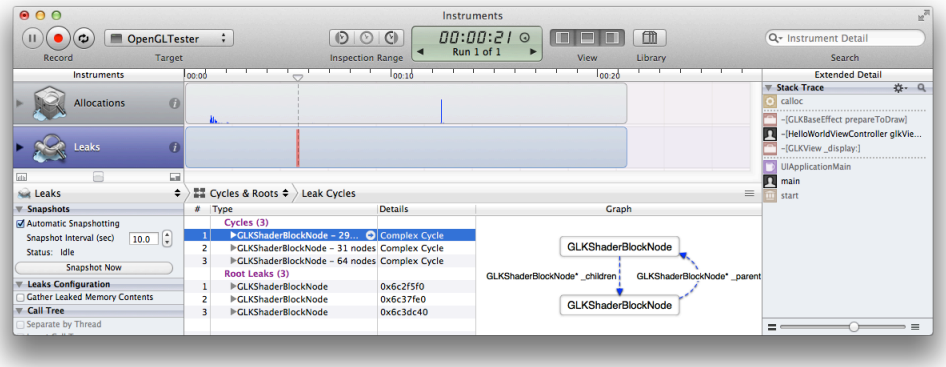


Note: If a leak isn't an object, you may be calling an API that assumes ownership of a `malloc`-created memory block for which you are missing a corresponding call to free().

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. It provides a graph of the reference cycle that is causing the leak.

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.



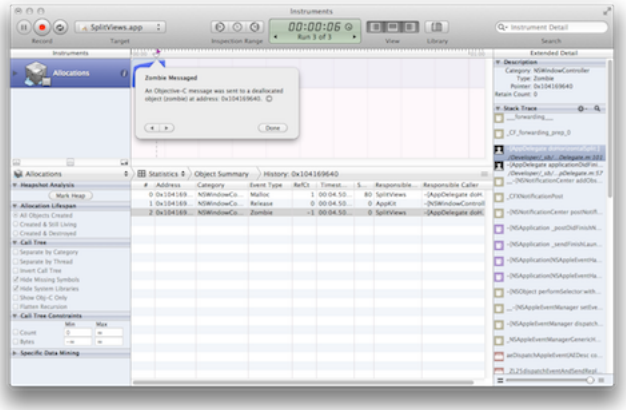
Eradicating Zombies with the Zombies Trace Template

The Zombies trace template measures general memory usage while focusing on the detection of overrelease "zombie" objects. It also provides statistics on object allocations by class as well as memory address histories for all active allocations. It consists of the Allocations instrument only.

The Zombies template substitutes 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.

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 pane and double-click the zombie event type in the object history table.
6. In the stack trace that appears, double-click Responsible Caller to display the responsible code.



Tip: The Zombies template causes memory growth because the zombies are never deallocated. So for iOS apps, use it with 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

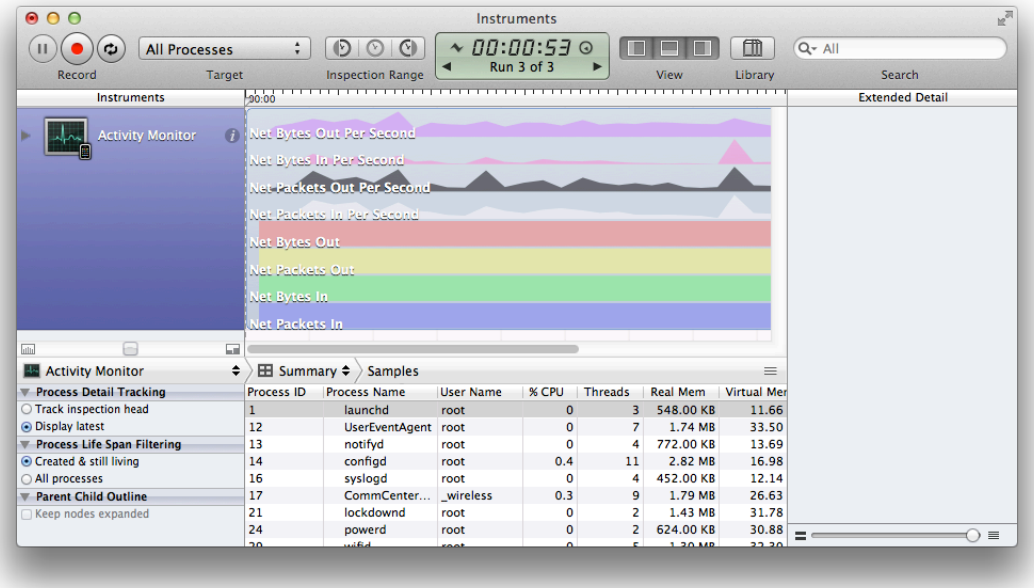
Apps can be complicated programs with a lot of information being passed between the device and the user. The I/O Activity trace template in Instruments help you see what your app is doing and where it is sending and receiving information. This chapter shows you how to use these trace templates and monitor 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 also monitors all existing processes and can be used to attach new instruments to specific processes, monitor parent-child process hierarchies, and quit running processes. The Activity Monitor trace template consists of the Activity Monitor instrument.

Use the network statistics in the Activity Monitor instrument to see which processes are sending and receiving information. Examine this information 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



The Trace Highlights view option does not provide any useful graphs when looking at network connections. To provide useful information, the Activity Monitor instrument must be configured. The following configuration options provide network specific information through the Activity Monitor. For statistic definitions and complete configuration options, see *Instruments User Reference*.

- 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

Analyzing Network Connections with the Connection Trace Template

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

To view network connections used by your app

1. Connect your iOS device.

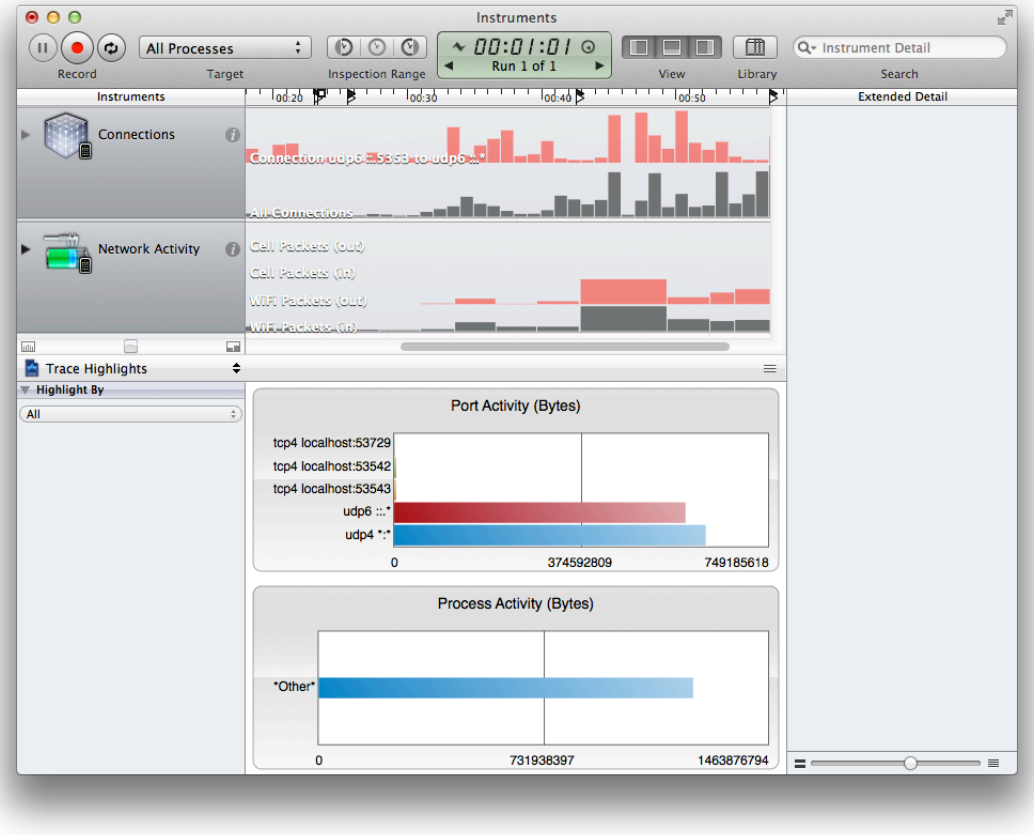
   You can use a physical or wireless connection. See "To enable a wireless device" (page 35).

2. Choose a target from the Target pop-up menu.
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 amount of bytes used other processes. Switching to the Connection view shows the collected information in column form.

You can compare the amount of data passed by different connections using the Connections instrument (see Figure 8-2). Clicking a checkbox in the Graph column next to a connection displays that information in the track pane. A graph of all connections is always displayed in the track pane.

Figure 8-2 Viewing network connections



In conjunction with the Connections instrument, the Network Activity instrument measures the number of packets that are sent and received by your app.

Measuring Graphics Performance in Your iOS Device

Extensive use of graphics in your 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 three trace templates found in the iOS Graphics section to profile your app. Ensure that the frame rate is high enough and that your graphics don't impede your app's performance.

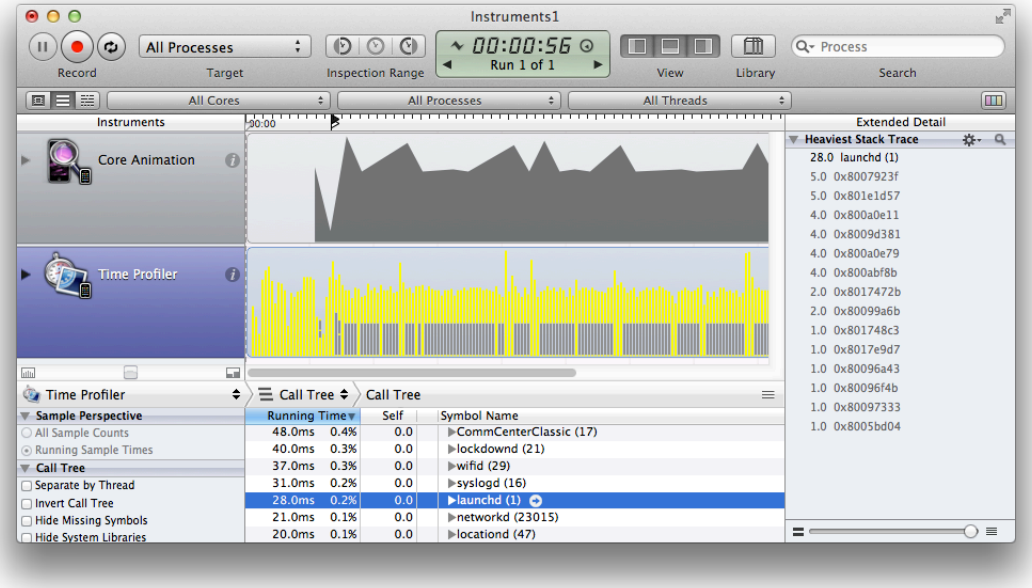
Measuring Core Animation Graphics Performance

Instruments uses the Core Animation instrument to measure your app's graphical performance on iOS devices. The Core Animation trace template provides a quick and lightweight option for measuring the number of frames per second rendered by your app. This instrument is not intended to be used to measure OpenGL ES performance. You can 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 identify areas of your code that need to be improved.

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



Core Animation contains several useful debugging options in the Detail pane. You do not need to be running a trace in order to see these options working on your iOS device. Select the running process from the Target pop-up menu.

- **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.

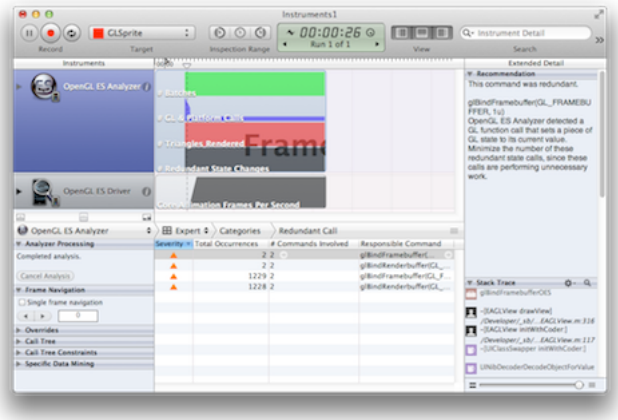
- **Flash Updated Regions.** Colors regions on your iOS device in yellow when that region is updated by the graphics processor.

Measuring OpenGL Activity 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 recommendation for addressing found problems. It consists of the OpenGL ES Analyzer and the OpenGL ES Driver instruments.

To get OpenGL ES Analyzer to make suggestions for your app

1. Open the OpenGL ES Analysis template in the iOS group.
2. Click the Choose Target pop-up and select your iOS device.
3. Click the Choose Target pop-up a second time and choose the app you want to analyze.
4. Click the Record button to begin recording data, and exercise your OpenGL graphics code.
5. Click the Stop button when issues stop accumulating in the detail pane.

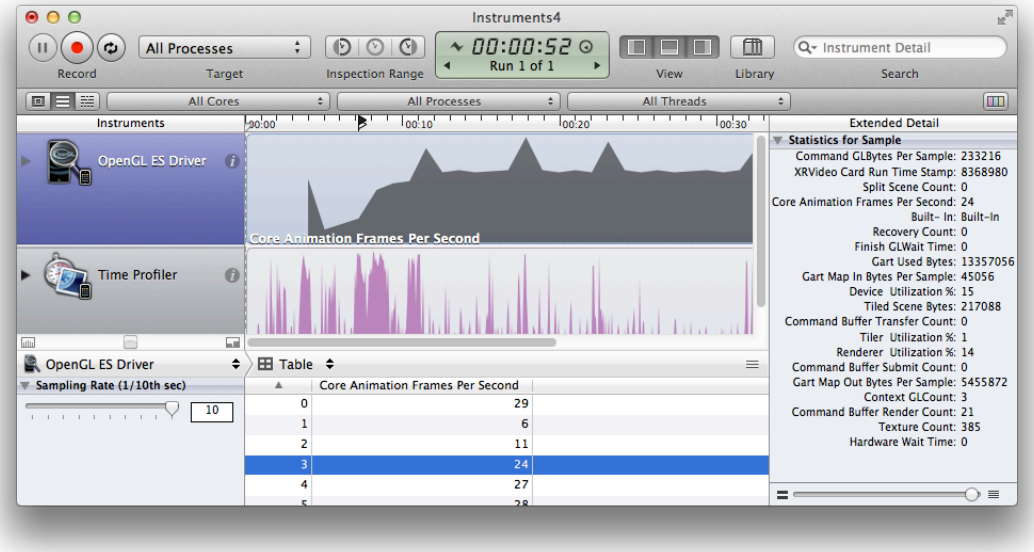


Errors are listed in the Detail pane, showing total occurrences, unique occurrences, category, summary, and (at the far left) a severity code that is either a red square for most severe, or an orange triangle for less severe. When an error is selected a recommendation is provided in the Extended Detail pane on how to fix the detected problem.

Finding Bottlenecks with the OpenGL ES Driver Trace Template

The OpenGL ES 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 Extended Detail pane displays all of the gathered information for a specific sample. Each statistic can also be displayed in the track pane by configuring the OpenGL ES Driver to display that particular statistic. For detailed information about the statistic offered, see "OpenGL ES Driver Instrument" in the *Instruments User Reference*.

Figure 9-2 Detailed information for a Core Animation sample



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 your CPU. Effective use of CPUs 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 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 identify 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

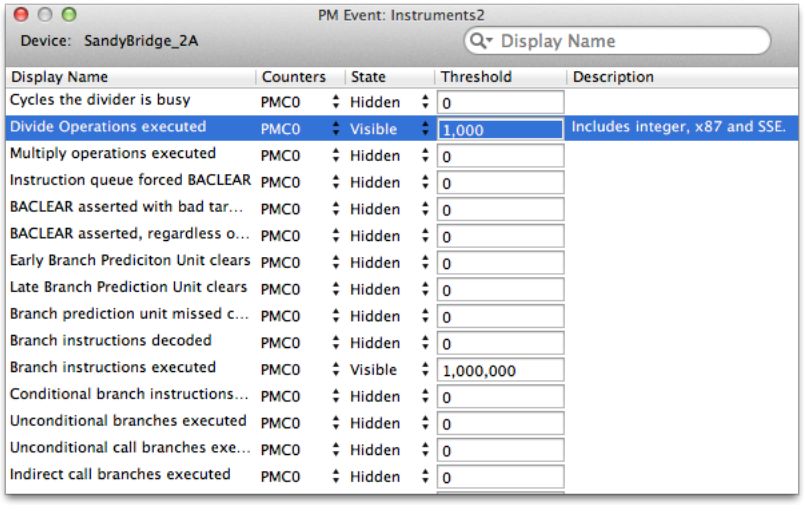
1. Open the Counters trace template.
2. Click the Inspector button in the Counters instrument.
3. Click the plus button (+).
4. Click the event to change to another event.

The PMC events list is populated with an initial set of common events. You can add events specific to your app through the Window menu.

To add new PMC events

1. Select Window > Manage PM Events.

2. Click the state corresponding to the desired event and select Visible or Favorite.



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.

If you plan on recording the same PMC events frequently, save them in a 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 54).

Tracking a Single Event

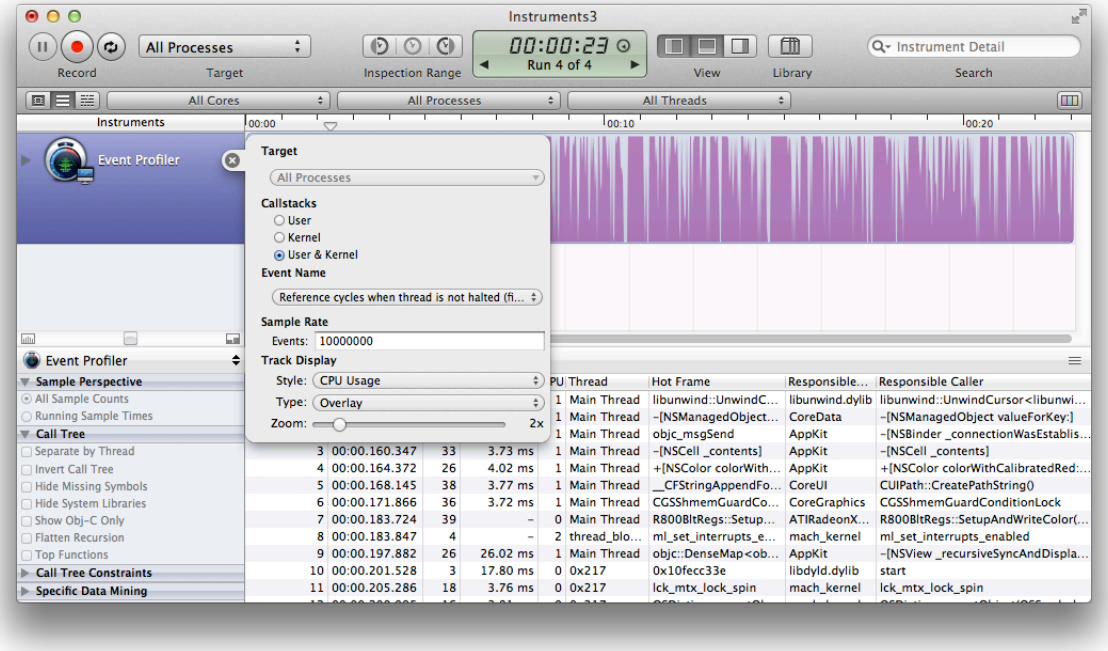
The Event Profiler instrument tracks performance monitor interrupt (PMI) events, but in this case, only one event is tracked and you can get more detail about it. You set the size of each sample set collected by the Event Profiler instrument. Event Profiler then provides you with information about how active the tracked PMI event was during the sample. Use the track pane slider to identify those samples showing a lot of activity. After you have identified high activity areas, use the Detail pane to get more information on each particular example.

To add new PMI events

1. Choose Window > Manage PM Events.
2. Click the state corresponding to that event and select Visible or Favorite.
3. In the Threshold field, enter the number of times an event must occur before a sample is taken.

Choose the PMI event you want to track by selecting it from the Event Name pop-up menu control. If the required event is not listed, make sure that you have enabled it. See "To add new PMC events" (page 70) to create a new event. Figure 10-1 shows the Event Profiler tracking a single PMC event.

Figure 10-1 Tracking one PMI event with the Event Profiler instrument



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.

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.

The Developer Setting appears only after the 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. Turn on developer logging in 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.

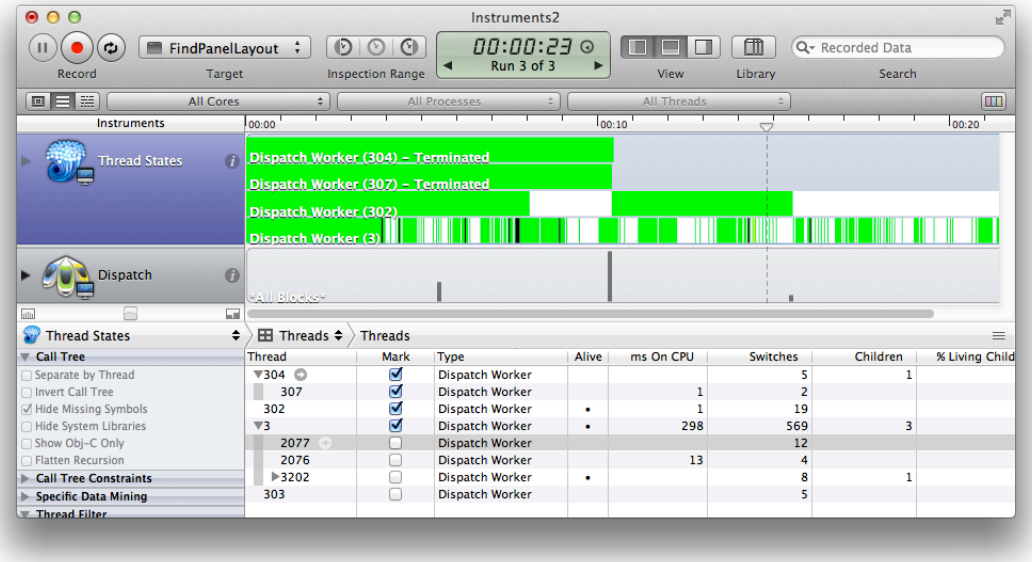
Minimize the amount of energy your app uses by ensuring that you turn off any radios that you don't actively need. You can verify if you have turned off a particular radio using the Energy Diagnostics trace template. Each radio is depicted with red in the track pane to designate that it is on and black to designate that it is turned off.

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 threads 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 four threads being tracked.

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



To view thread usage in your app

1. Select the Multicore trace template.
2. Run your app.
3. Choose the threads to examine by selecting the checkbox in the Mark column in the Detail pane.

The following actions are captured in the track pane. Color notations are the default color for each action, 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 75).

Delving into Core Usage with the Time Profiler Trace Template

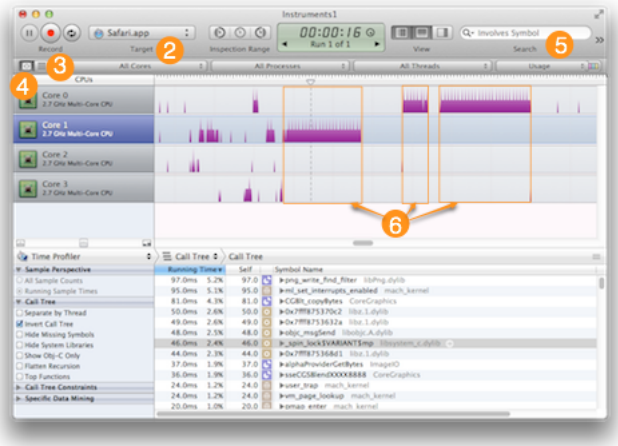
The Time Profiler trace template performs low-overhead, time-based sampling of processes running on the system's CPUs. It consists of the Time Profiler instrument only.

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. The CPU strategy Usage view facilitates comparison of 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 instrument.
2. Choose your app from the Choose Target pop-up menu.
3. Click Record, exercise your app, and then click Stop to capture data.
4. Click the CPU strategy button.
5. Select Usage.

6. 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.

Monitoring OS X App Activity

Monitor your app to find problems with it before you release it to the world. To monitor your app in OS X, you profile it with the file system and behavior trace templates supplied by Instruments. Using these templates, you manipulate your app and watch as your code reacts to the manipulation. For example, you can make sure your app doesn't suddenly terminate, make sure it writes to approved directories only, and then have Instruments run it for you.

Tracking File System Usage

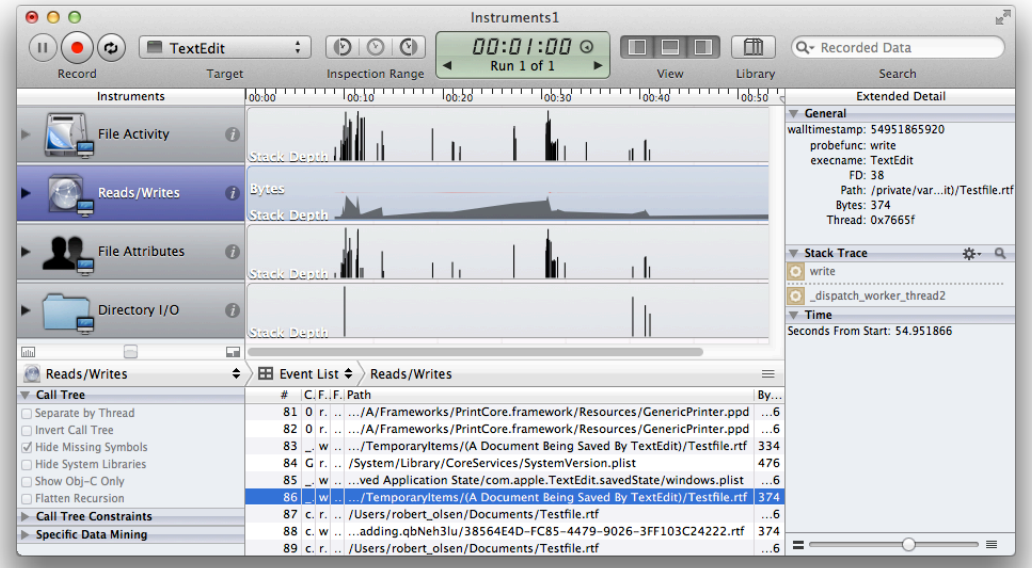
The File Activity trace template monitors file and directory activity, including file open/close calls, file permission modifications, directory creation, and file moves. It consists of the File Activity, Reads/Writes, File Attributes, and Directory I/O instruments.

Every app submitted to the App Store is required to save information in specific directories unless the user explicitly chooses to save the information in another location. Apps that do not save their information in the correct directories are rejected during submission. Use the File Activity trace template to ensure that your app follows the rules found in "File-System Usage Requirements for the Mac App Store" in *Submitting to the Mac App Store*.

To see where you app saves information

1. Select the File Activity trace template.
2. Profile your app.
3. Select the Reads/Writes instrument.
4. Look for `write` calls in the Function column.

5. Verify that `write` calls follow the rules stated in "File-System Usage Requirements for the Mac App Store" in *Submitting to the Mac App Store*.



Identifying Layout Changes with the Cocoa Layout Trace Template

To properly profile your app, you must identify what your app is doing when layout changes occur. The Cocoa Layout trace template accomplishes this by observing the changes to `NSLayoutConstraint` objects in order to help you debug your code. When you run your app, if the layout is not acting the way you expect, then chances are there is an error in your layout. The Cocoa Layout instrument makes it easy to identify these errors by taking a snapshot whenever the layout changes and providing you with information about what is happening.

To locate cocoa layout calls

1. Profile your app from inside of Xcode.

   See "To run Instruments while building your code" (page 11).

2. Exercise your app.
3. In the Detail pane, highlight the item that is not working as intended.
4. Pull up your code in the Detail pane's Console area.

   See "To see your personal code" (page 51).

Preventing Sudden Termination

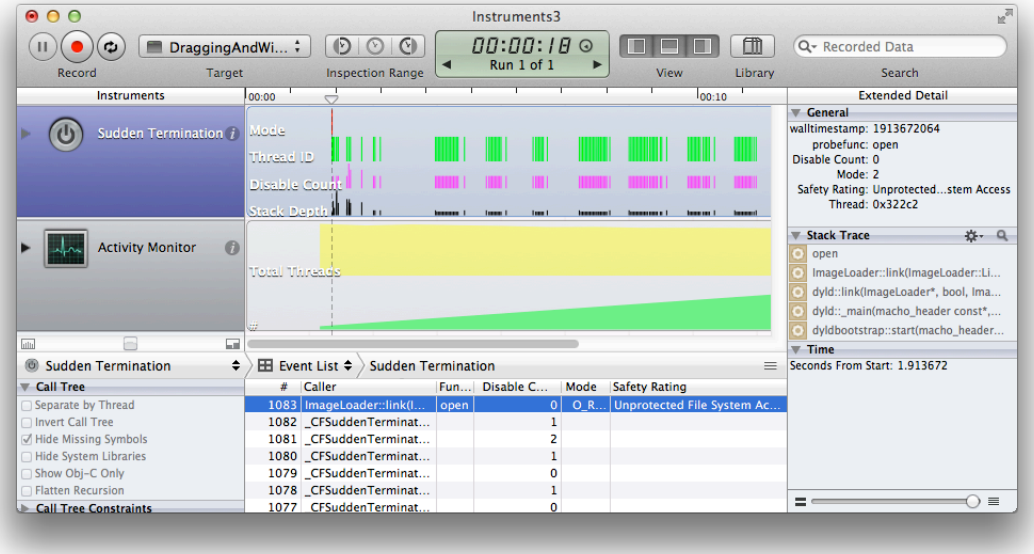
Preventing your app from suddenly terminating should be a high priority when creating your app because no person enjoys using an app that has a tendency to shut down while in use. Often, sudden termination is caused when the file system is accessed and the system is not protected by calls designed to prevent the sudden termination of the app.

To find unprotected file access

1. Select the Sudden Termination trace template.
2. Record your data.
3. In the Detail pane, sort by Function, Mode, or Safety Rating.

After you have identified an unprotected file access, select the section in the Detail pane to bring up more information in the Extended Detail pane. Here you can see the stack trace at the time of the access and know what thread was running when it happened. The Activity Monitor instrument allows you to see more detailed information about what was happening at the time of the call. Figure 11-1 shows Instruments finding unprotected file access calls.

Figure 11-1 Finding an unprotected file access call



Using Instruments to Run Your App

The UI Recorder trace template records the user interface events of a process launched from Instruments. You can then play this recording back in order to drive the user interface of the target process while other instruments record new data. It consists of the User Interface instrument only.

To properly test an app, you may need to interact with it multiple times. Using the same actions each time you test provides you with a consistent testing platform. You can then compare your testing results reliably to previous results to ensure that any changes made to your code have not adversely affected the app.

To collect data using the same set of actions each time

1. Select the User Interface instrument.
2. Click Record to begin gathering data.
3. Perform the events you want to record.
4. Click Stop after the data has been recorded.
5. Choose File > Save to save the recording.

See the HTML version of this document to view the video.

User interface events are color coded according to the type of event:

- **Blue.** Mouse events
- **Green.** Key events
- **Yellow.** System events

Note: Your system must be set up to use assistive devices. Select "Enable access for assistive devices" in System Preferences > Universal Access to enable this feature.

Automating UI Testing

Use the Automation instrument to automate user interface tests for your iOS app through test scripts that you write. These scripts simulate user actions by calling UI Automation, a JavaScript programming interface that specifies actions to be performed in your app as it runs. During the tests, the system returns log information to you.

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

This chapter describes how you use the Automation template in Instruments to execute scripts. It also describes how to mesh your scripts with the UI Automation programming interface 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

As you work through this chapter, look for more detailed information about each class in *UI Automation JavaScript Reference*.

An important benefit of the Automation instrument is that you can use it with other instruments to perform sophisticated tests such as tracking down memory leaks and isolating causes of performance problems.

Note: The Automation instrument only works with apps that have been code signed with a development provisioning profile. Apps signed with a distribution provisioning profile can not be automated with the UI Automation programming interface.

Important: Simulated actions may not prevent the test device from auto-locking. Before running tests on a device, you should set the Auto-Lock preference to Never.

Writing an Automation Test Script

Your test script must be a valid executable JavaScript file that is accessible to the instrument on the host computer. Because the script runs outside of your app, the app version you are testing can be the one you submit to the App Store.

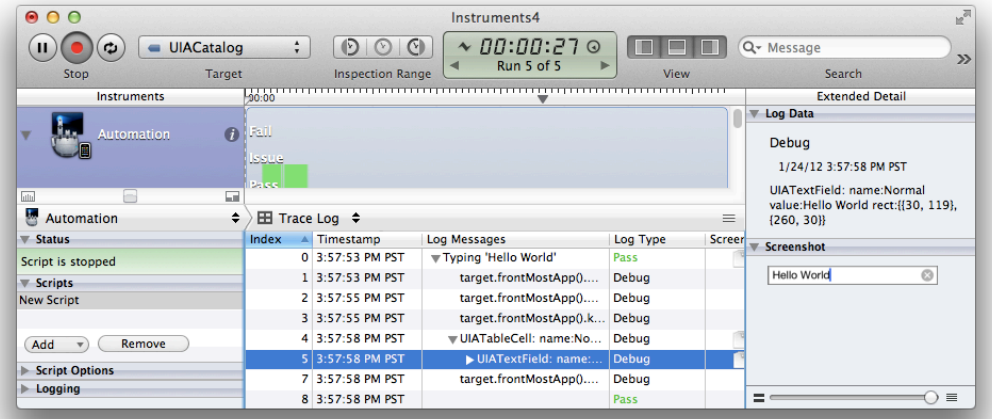
Because each app is different, often a script for one app is not acceptable for use in another app. In this case, you can create your own script inside of Instruments.

To create a script

1. Select the Automation trace template.
2. Click Add > Create.
3. Double-click New Script to change the name of the script.
4. In the Detail pane, select Console to enter the code for your script.
5. Choose a target for your script.
6. Click the Play button at the bottom of the Console.

After you create a script, you will want to use it throughout the development of your app. You do this by importing your saved script and running it with the Automation instrument. Figure 12-1 shows a completed run using a script.

Figure 12-1 Running an iOS app through scripting



Testing Your Automation Script

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.

Your test script must be a valid executable JavaScript file accessible to the instrument on the host computer. It runs outside your app, so the tested version of your app can be the same version that you submit to the iTunes App store.

You can create as many scripts as you like, but you can run only one at a time. The API does offer a `#import` directive that allows you to write smaller, reusable discrete test scripts. For example, if you were to define commonly used functions in a file named `TestUtilities.js`, you could make those functions available for use in your test script by including in that script the line

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

To import a previously saved script

1. Select the Automation trace template.
2. Click Add > Import.
3. Navigate to your saved script file and click Open.

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.

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.

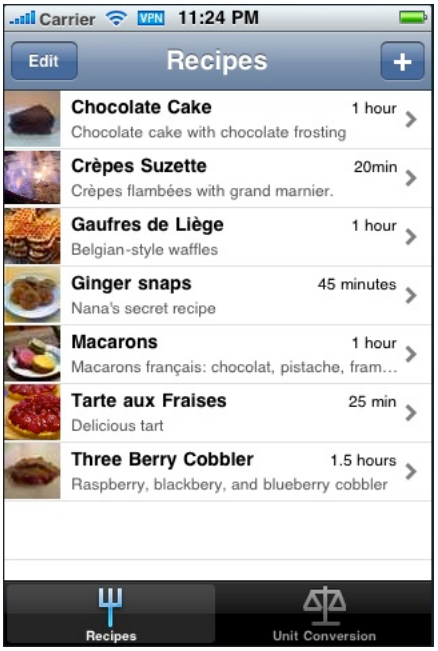
> **Important:** You must explicitly stop recording. Completion or termination of your script does not turn off recording.

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 12-2, which is available as the code sample *iPhoneCoreDataRecipes* from the iOS Dev Center.

Figure 12-2 The Recipes app (Recipes screen)

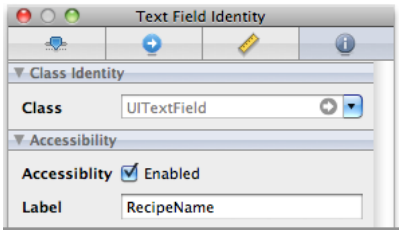


UI Element Accessibility

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 12-3.

Figure 12-3 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();
```

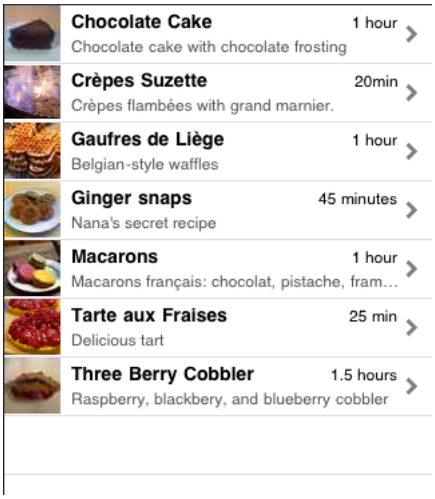
To reach the app window, the main window of your app, you would specify

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

At startup, the Recipes app window appears as shown in Figure 12-2 (page 84).

Inside the window, the recipe list is presented in an individual view, in this case, a table view, see Figure 12-4 (page 86).

Figure 12-4 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:

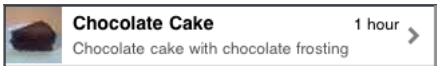
```
UIATarget.localTarget().frontMostApp().mainWindow().tableViews()[0];
```

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"];
```



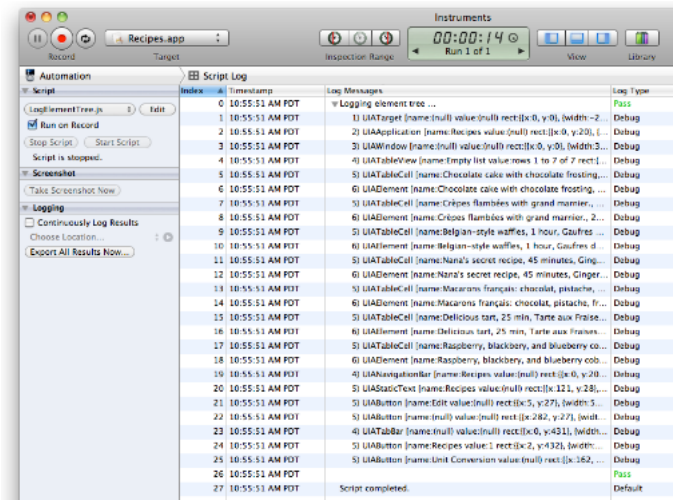
Displaying the Element Hierarchy

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();
```

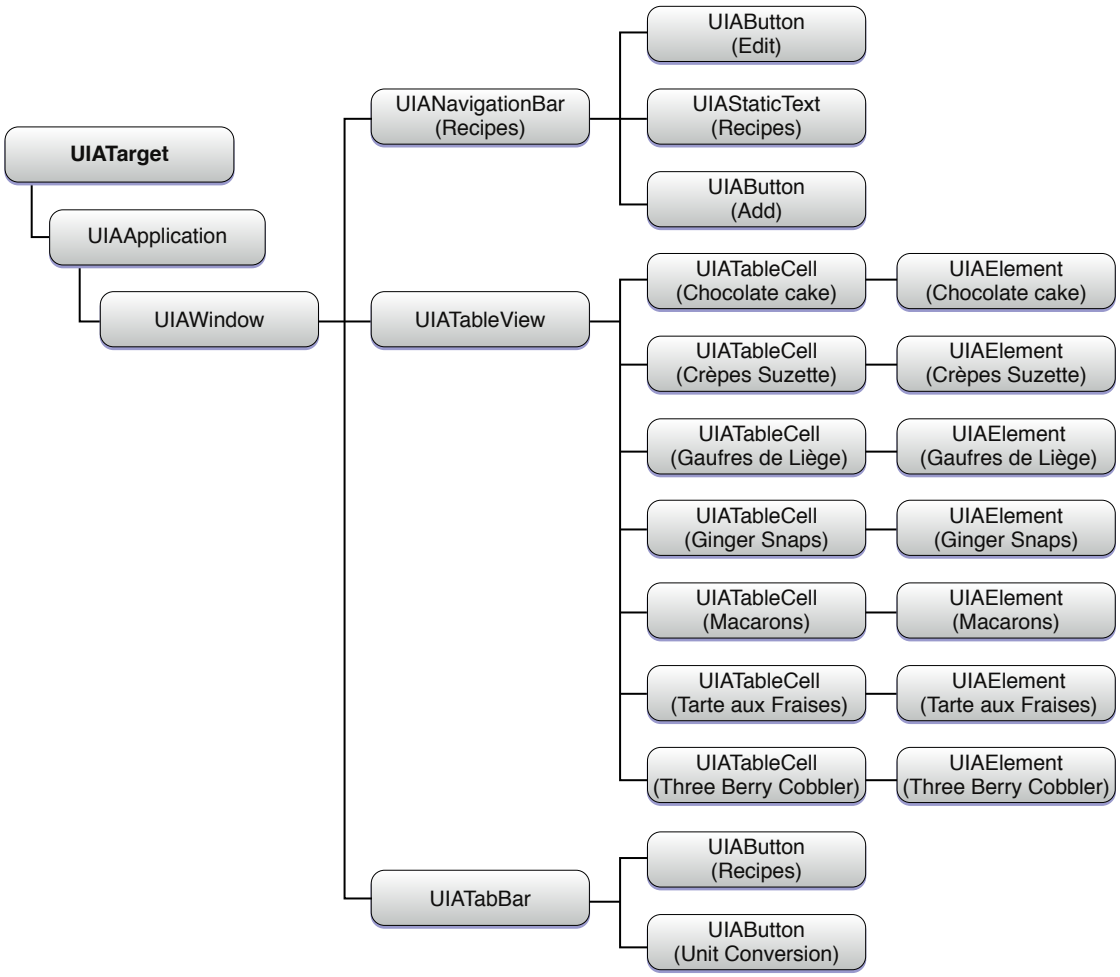
The output of the command is captured in the log displayed by the Automation instrument, as in Figure 12-5.

Figure 12-5 Output from logElementTree method



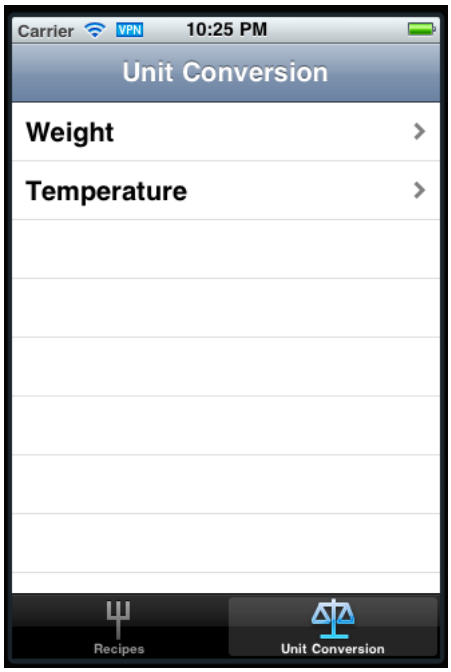
Note the number at the beginning of each element line item, indicating that element's level in the hierarchy. These levels may be viewed conceptually, as in Figure 12-6.

Figure 12-6 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 12-7.

Figure 12-7 Recipes app (Unit Conversion screen)

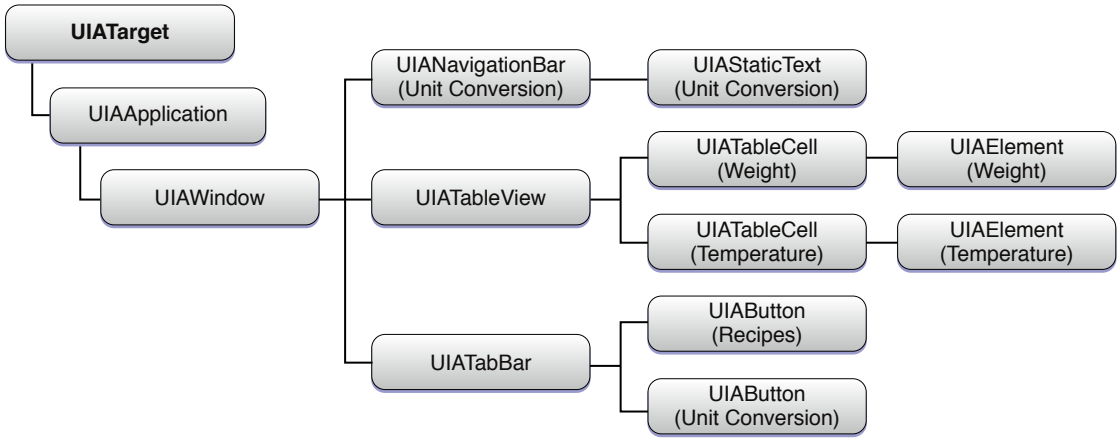


The following code taps the Unit Conversion tab in the tab bar to display the associated screen and then logs the element hierarchy associated with it:

```
// 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 12-8. 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 12-8 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 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*.

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 will be 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.

Adding Timing Flexibility with Timeout Periods

Your script may need to wait for some action to complete. 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 will repeatedly attempt 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. The default timeout period is five seconds, but your script can change that at any time.

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);
```

Logging Test Results and Data

Your script reports log information to the Automation instrument, which gathers it and reports it for your 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 in Instruments with the specified filename (`SS001-2_AddedIngredient`, in this case).

> **Note:** Screenshots are currently not supported when targeting iOS Simulator. A screenshot capture attempt is, however, indicated by a log message indicating a failed attempt.

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. Even worse, a telemarketing autodialer could pick the number for your phone just as you launch your script.

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 to log a message that this type of alert happened 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 12-1.

> **Note:** As regards 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 12-1 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 12-2.

Table 12-2 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.

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 v10.5. Because DTrace taps into the operating system kernel, you have access to low-level operation about the kernel itself and about 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 a custom instrument 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 OpenSolaris website. For information about the `dtrace` command-line tool, see the `dtrace` man page.

Note: Several Apple apps—namely, iTunes, DVD Player, Front Row, and apps that use QuickTime—prevent the collection of data through DTrace (either temporarily or permanently) in order to protect sensitive 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 lets you 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 the 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.

An 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

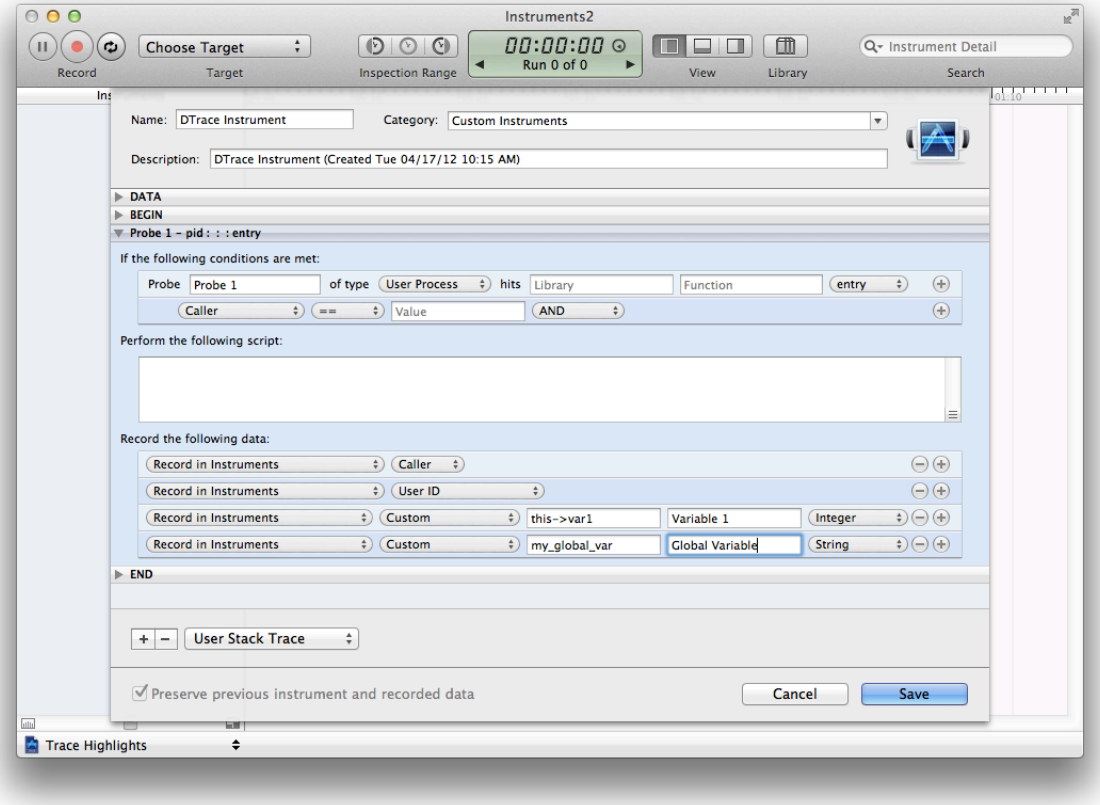
All instruments must have at least one probe with its associated actions. Similarly, all instruments should have an appropriate name and description to identify them to Instruments users. Instruments displays your instrument's descriptive information in the library window. Providing good information 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 111).

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 13-1. You use this sheet to specify your instrument information, including any probes and custom scripts.

Figure 13-1 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 104).
- **Probe action.** The data to record or the script to execute when your probe fires; see "Adding Actions to a Probe" (page 109).

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 plus (+) 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 minus (-) 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 13-1 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 13-1 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.

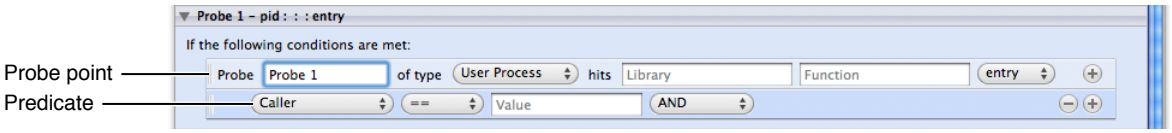
Adding Predicates to a Probe

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 plus (+) button.
2. Select the type of predicate.
3. Define the predicate values.

Figure 13-2 Adding a predicate



You can add subsequent predicates using the plus buttons of either the probe or the predicate. To remove a predicate, click the minus button next to the predicate. To rearrange predicates, click the predicate line and drag it to a new location.

Instruments evaluates predicates from top to bottom in the order in which they appear. 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 line to choose the data to inspect as part of the condition. Table 13-2 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 13-2 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.
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.

Variable	DTrace variable	Description
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.
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.

Variable	DTrace variable	Description
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.

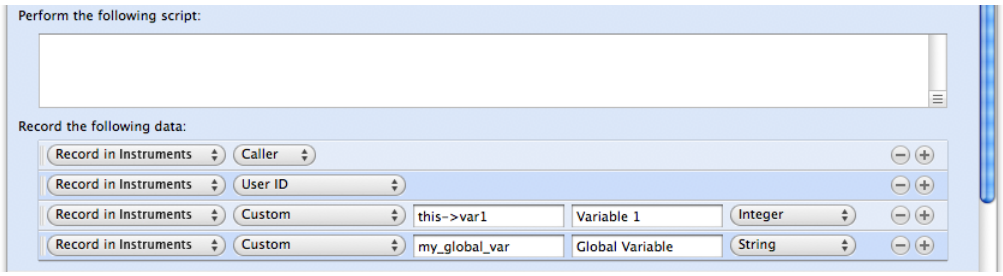
Adding Actions to a Probe

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 13-3 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 13-3 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 returns 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 pane. 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 13-4.

Figure 13-4 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 13-2 (page 107). For more information on scripts and script variables, see "Tips for Writing Custom Scripts" (page 111). For more information on DTrace subroutines, including the `copyin` and `copyinstr` subroutines, see the *Solaris Dynamic Tracing Guide*, available from the OpenSolaris website.

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 on the OpenSolaris website.

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 13-1 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 13-1 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 13-3 lists the scoping levels for variables and the syntax for using variables at each level.

Table 13-3 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. For information about whether an instrument is DTrace-based, refer to that instrument in *Instruments User Reference* .

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.

Document Revision History

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

Date	Notes
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”.
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.

Date	Notes
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