Instruments User Guide

Tools & Languages: Performance Analysis Tools

2011-05-07

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

Contents

Figures, Tables, and Listings

Introduction to Instruments User Guide

Instruments is an application for dynamically tracing and profiling Mac 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 programs and the operating system.

With the Instruments application, you use special tools (known as *instruments*) to trace different aspects of a process' behavior. You can also use the application to record a sequence of user interface actions and replay them, using one or more instruments to gather data.

The Instruments application 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 application 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 program
- Automate testing of your code
- Stress-test parts of your application
- Perform general system-level troubleshooting
- Gain a deeper understanding of how your code works

Instruments is available with Xcode 3.0 and Mac OS X 10.5 and later.

This document describes the Instruments user interface and gives an overview of how to use the Instruments application to trace processes and view data. It is intended for developers and system administrators who want to use Instruments to better understand the behavior of their programs or of the system as a whole.

Organization of This Document

The following chapters describe how to use the Instruments application:

- "Instruments Quick Start" (page 11) gives a brief overview of the Instruments application and introduces the main window.
- "Adding and Configuring Instruments" (page 19) describes how to add and configure instruments and run them to collect data on one or more processes. This chapter also shows how to choose the program to trace.
- "Recording Trace Data" (page 31) describes the ways you can initiate traces and gather trace data.
- "Working with a User Interface Track" (page 39) describes how to record and replay a sequence of user actions.
- "Viewing and Analyzing Trace Data" (page 43) describes the tools you use to view the data returned by the instruments.
- "Saving and Importing Trace Data" (page 63) describes how you save trace documents and data and how you import data from other sources.
- "Creating Custom Instruments with DTrace" (page 65) shows how to create and configure your own DTrace-based custom instrument.
- "Built-in Instruments" (page 77) describes the instruments that ship with Instruments.

Getting Xcode

Instruments is part of Xcode, a comprehensive suite of developer tools for creating Mac OS X and iOS software. Xcode includes applications to help you design, create, debug, and optimize your software. Xcode also includes header files, sample code, and documentation for Apple technologies. You can download Xcode from Apple's Developer Tools website. Registration is required but free.

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, previously you had to sample a program and analyze its memory behavior during two separate execution runs. In Instruments, you can perform these tasks at the same time. You can then use the resulting data to spot trends between the code being run by your program and its corresponding memory usage.

The Instruments application uses **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 applications—namely iTunes, DVD Player, and Front Row, and applications 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 applications when performing systemwide data collection.

All work in Instruments is done in a trace document. A **trace document** collects the data gathered by the instruments associated with that document. 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 towards gathering trace data, one of the most sophisticated instruments helps automate the collection of data. Using the User Interface instrument, 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 available as part of the Xcode Tools installation. There are several ways to launch Instruments:

- You can launch Instruments by double-clicking its application icon in the Finder. The Instruments application is located in *<Xcode>*`/Applications`, where *<Xcode>* is the root directory of your Xcode installation. (The default root directory for an Xcode installation is the `/Developer` directory.)

- You can use Xcode to launch Instruments and target an executable in your project. Choose an Instruments template from the Run > Start with Performance Tool menu in Xcode. Instruments launches with the specified template ready to run the current executable, as configured in Xcode.

  Note: When launching your program from an Instruments template in this manner, you must explicitly stop your executable—either from the program or from Instruments—after tracing is complete. This does not close the Instruments trace document. Instead, if you relaunch the executable, Instruments displays the new session alongside the previous one, as shown in Figure 4-2 (page 41). This lets you change your code in Xcode, rebuild, run, and compare the resulting trace data to the data generated before the change.

- You can launch Instruments by double-clicking an Instruments template or trace document. See "Saving and Importing Trace Data" (page 63) to learn how to save an instrument configuration or user interface recording as a trace template document.

Creating a Trace Document

When you launch Instruments, the application automatically creates a new document for you. You can also create new documents by choosing File > New.

For each new document you create, Instruments prompts you to select a starting template. These templates define the initial set of instruments you plan to use in your trace document. Instruments provides several different templates, listed in Table 1-1, each geared toward different goals. You can use the Blank template if you want to add a specific set of instruments to your trace document manually. For any of the templates, you can also remove instruments you do not want.

Table 1-1 Instruments starting templates

Template	Type	Description
Blank	Mac OS X, iOS, iOS Simulator, User	Creates an empty trace document to which you can add your own combination of instruments. To learn how to select and add instruments, see "Adding and Configuring Instruments" (page 19). For descriptions of the individual built-in instruments, see "Built-in Instruments" (page 77).
Activity Monitor	Mac OS X, iOS, iOS Simulator	Adds the Activity Monitor instrument to your document. Use this template if you want to correlate the system workload with the virtual memory size.
Allocations	Mac OS X, iOS, iOS Simulator	Adds the Allocations and VM Tracker instruments to your document. Use this template to monitor memory and object-allocation patterns in your program. (To use this template, you must launch your process from Instruments.)
Automation	iOS, iOS Simulator	Adds the Automation instrument to your document. Use this template to automate user interface tests of your iOS application.

Template	Type	Description
Core Animation	iOS	Adds the Core Animation and Sampler instruments to your document. Use this template to measure the number of Core Animation frames per second in a process running on an iOS device, and to see visual hints that help you understand how content is rendered on the screen.
Core Data	Mac OS X	Adds the Core Data Fetches, Core Data Cache Misses, and Core Data Saves instruments to your document. Use this template to monitor data store interactions in a Core Data application.
CPU Sampler	Mac OS X, iOS, iOS Simulator	Adds the Sampler and CPU Monitor instruments to your document. Use this template if you want to correlate the overall system workload with the work being done specifically by your application.
Dispatch	Mac OS X	Adds the Dispatch instrument to your document. Use this template if you want to capture information about GCD queues created by your application and about the block objects executing on these queues.
Energy Diagnostics	iOS	Adds the Energy Diagnostics, CPU Activity, Display Brightness, Sleep/Wake, Bluetooth, WiFi, and GPS instruments to your document. Use this template to get diagnostic information regarding energy usage in iOS devices.
File Activity	Mac OS X, iOS Simulator	Adds the File Activity, Reads/Writes, File Attributes, and Directory I/O instruments to your document. Use this template if you want to examine file usage patterns in the system. This combination of instruments monitors open, close, read, and write operations on files and also monitors changes in the file system itself, including permission and owner changes.
GC Monitor	Mac OS X	Adds the ObjectGraph, Allocations, and Garbage Collection instruments to your document. Use this template to measure the data reclaimed in the scavenge phase of the garbage collector.
Leaks	Mac OS X, iOS, iOS Simulator	Adds the Allocations and Leaks instruments to your document. Use this template to monitor memory usage in your application and to detect memory leaks. (To use this template, you must launch your process from Instruments.)
Multicore	Mac OS X	Adds the Thread States and Dispatch instruments to your document. Use this template to analyze multicore performance, including thread state, dispatch queues, and block usage.
OpenGL ES Analysis	iOS	Adds the OpenGL ES Analyzer and OpenGL ES Driver instruments to your document. Use this template to measure OpenGL ES activity and get recommendations for addressing problems.
OpenGL ES Driver	iOS	Adds the OpenGL ES Driver and Sampler instruments to your document. Use this template to determine how efficiently you're using OpenGL and the GPU on iOS devices.

Template	Type	Description
Sudden Termination	Mac OS X	Adds the Sudden Termination and Activity Monitor instruments to your document. Use this template to analyze sudden termination support. It reports unprotected file-system access the target process should be, but is not, guarding with calls to disable sudden termination. It also provides activity monitoring across all processes, including sudden termination status for each.
System Usage	iOS	Adds the I/O Activity instrument to your document. Use this template to record calls to functions that operate on files in a process running on an iOS device.
Threads	Mac OS X, iOS Simulator	Adds the Thread States instrument to your document. Use this template to analyze thread state transitions within a process, including running and terminated threads, thread state, and associated backtraces.
Time Profiler	Mac OS X, iOS, iOS Simulator	Adds the Time Profiler instrument to your document. Use this template to perform low-overhead time-based sampling of one or all processes.
UI Recorder	Mac OS X	Adds the User Interface instrument to your document. Use this template as a starting point for recording a series of user interactions with your application. You can use this feature to reproduce a series of events multiple times, gathering a new set of data during each successive run. You can then compare the sets of data knowing that the behavior that generated them was identical. Typically, you would start with this template and add additional instruments to your document to gather data.
Zombies	Mac OS X, iOS Simulator	Adds the Allocations instrument to your document. Use this template to measure general memory usage while focusing on the detection of over-released "zombie" objects.

If you do not want Instruments to ask you for a template when you create a new document, you can enable the Suppress template chooser option in the Instruments General preferences. For more information about adding and configuring the instruments in a document, see "Adding and Configuring Instruments" (page 19).

A Tour of 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 that you've gathered at both a high and low level.

Figure 1-1 shows a typical trace document. A trace document window presents a lot of information and therefore 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. Table 1-2 (page 15) provides descriptions for the key areas of this window.

Figure 1-1 The Instruments window while tracing

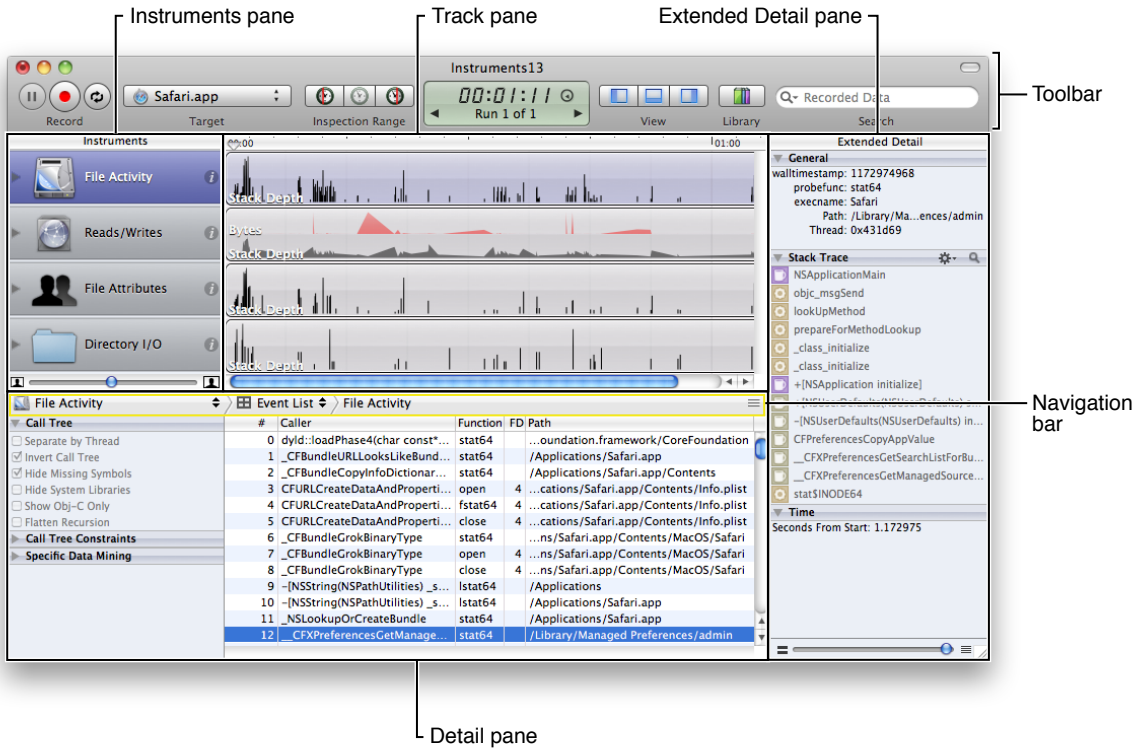


Table 1-2 lists some of the key features that are called out in Figure 1-1 (page 15) and provides a more in-depth discussion of how you use that feature.

Table 1-2 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. To learn more about instruments, see "Adding and Configuring Instruments" (page 19).
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. The information in this pane is read-only. You do use this pane to select specific data points you want to examine more closely, however. The track pane is described in more detail in "The Track Pane" (page 43).
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. For more information about this pane, see "The Detail Pane" (page 48).

Feature	Description
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. The Extended Detail pane is described in "The Extended Detail Pane" (page 51).
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 1-2 identifies the different controls on the toolbar, and Table 1-3 (page 16) provides detailed information about how you use those controls.

Figure 1-2 The Instruments toolbar

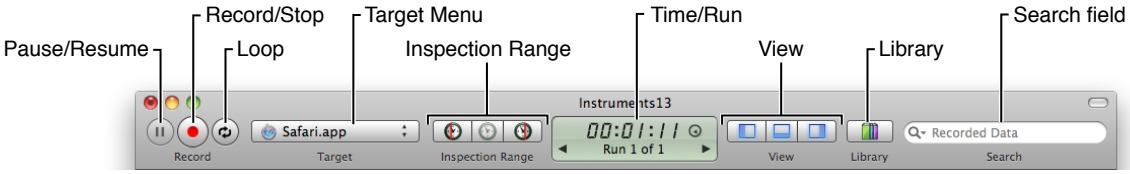


Table 1-3 Trace document toolbar controls

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. You use this button to begin gathering trace data. For more information, see "Collecting Data" (page 34).
Loop button	Sets whether the user interface recorder should loop during play back to repeat the recorded steps continuously. Use this to gather multiple runs of a given set of steps. For information about playing tracks, see "Replaying a User Interface Track" (page 41).
Target menu	Selects the trace target for the document. The **trace target** is the process (or processes) for which data is gathered. For more information on choosing a trace target, see "Choosing Which Process to Trace" (page 31).
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. For more information, see "Viewing Data for a Range of Time" (page 50).

Control	Description
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. For information about trace runs, see "Viewing Trace Runs" (page 44).
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 "Using the Instrument Library" (page 19).
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. For more information, see "Searching in the Detail Pane" (page 49).

Example: Performing a Quick Trace

To record trace data, specify the target from which you want to gather that data and press the Record button. Most instruments allow you to target a specific process on the system and some allow you to gather information for multiple processes.

The following steps show you how to create a new trace document, configure it, and record some data. The Instruments application should not be running prior to performing these steps.

1. Launch Instruments. The application automatically creates a new trace document and prompts you to select a template.
2. Select the Activity Monitor template and click the Choose button. Instruments adds the Activity Monitor instrument to the trace document.
3. In the Default Target menu of the trace document, choose All Processes.
4. Click the Record button.
5. Wait a few seconds so that Instruments can gather some data.
6. Click the Stop button.

Congratulations! You have just used Instruments to gather some trace data. Instruments displays several graphs related to the system load and virtual memory size in the track pane. The Detail pane shows the list of processes that were running during the data-gathering period. You can select the different view modes in the Detail pane to see the same data organized in different ways.

What's Next?

Now that you have been introduced to the basic concepts associated with Instruments, you can start exploring the application in more detail. The remaining chapters provide more in-depth coverage of Instruments features, including how to add and configure instruments, how to record a user interface track, how to analyze the data that you gather, and how to save the data you gather for later use.

Remember that program analysis is both an art and a science. On the scientific side, there are some guidelines that can be followed to find problems. For example, if your application has a large memory footprint, it is likely that the application will experience paging at some point, which leads to poor performance. The art comes in determining how to reduce the memory footprint because the answer for every application is different. Is the application caching too much data? Is it loading too many libraries at launch time and then not using them? Is it leaking memory? These are the questions you need to ask yourself, and Instruments is the tool you can use to find the answers.

Adding and Configuring Instruments

The Instruments application 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 configured to gather data from a different process on the system.

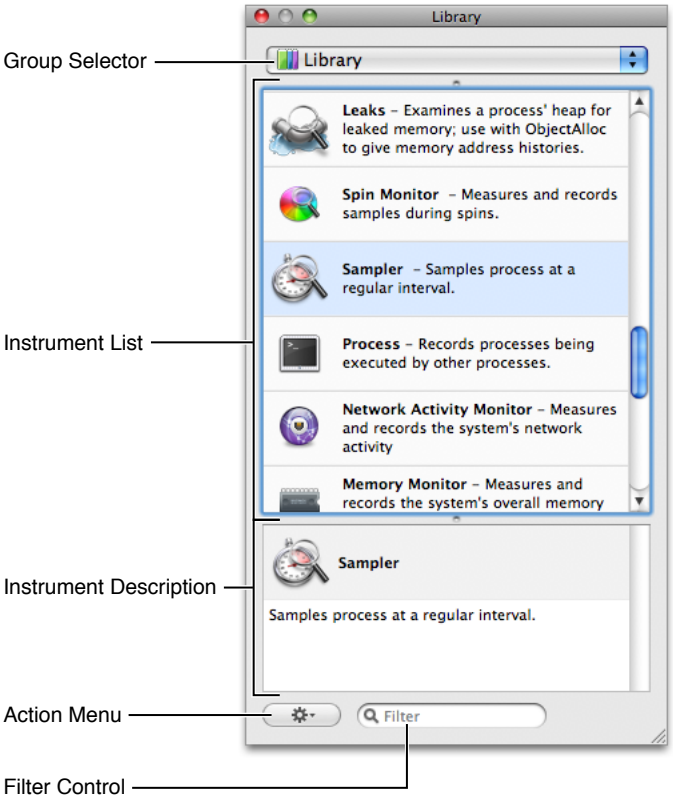
Instruments comes with a wide range of built-in instruments whose job is to gather specific data from one or more processes. Most of these instruments require little or no configuration to use. You simply add them to your trace document and begin gathering trace data. You can also create custom instruments, 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 with DTrace" (page 65).

Using the Instrument Library

The instrument library (shown in Figure 2-1) 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 already defined. To open this window, click the Library button in your trace document or choose Window > Library.

Figure 2-1 The instrument 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 selecting Show Group Banners from the Action menu. The Library window organizes instruments into groups 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.

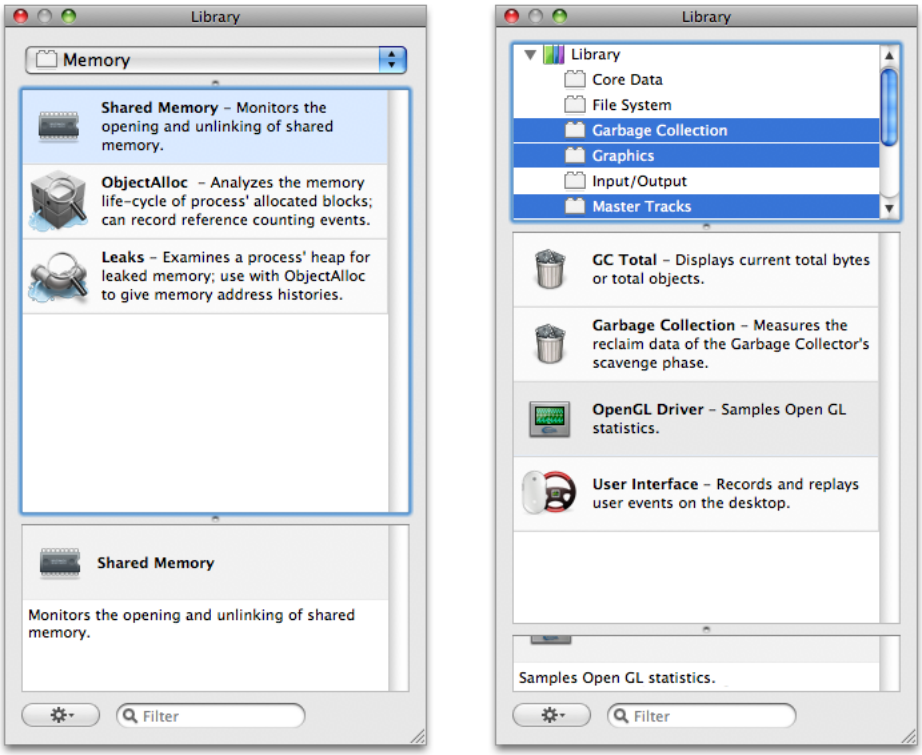
Finding an Instrument in the Library

By default, the Library window shows all of the available instruments. Each instrument, however, belongs to a larger group, which identifies the purpose of instrument and the type of data it collects. You can use the group selection controls at the top of the Library window to select one or more groups and thereby limit the number of instruments displayed by the Library window. 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.

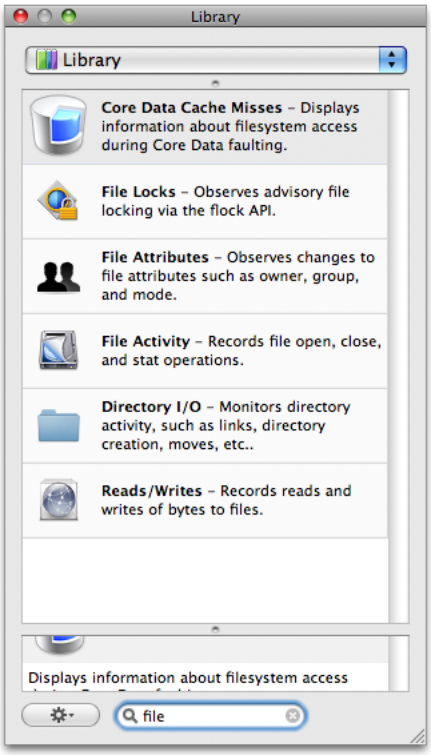
Figure 2-2 shows both standard mode and outline mode in the Library window. The window on the left shows standard mode, in which you select a single group using the pop-up menu. The window on the right shows the outline view, in which you can select multiple groups and manage your own custom groups.

Figure 2-2 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 2-3 shows instruments that match the search string `file`.

Figure 2-3 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.

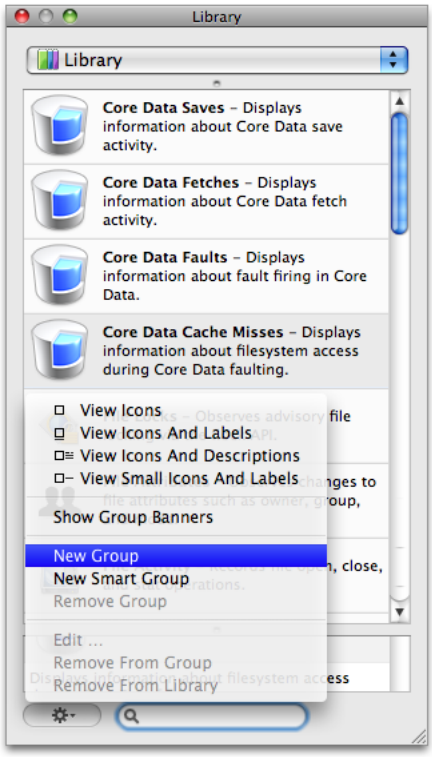
Creating Custom Instrument Groups

In addition to using the built-in instrument groups, you can create custom groups to organize instruments in a way that better suits your own use patterns. Instruments supports two different types of custom groups: static groups and smart groups. Static groups are just that—static. Instruments does not change their contents at all, leaving you free to configure the group however you want. Smart groups, on the other hand, are dynamic groups whose contents change automatically based on the criteria you specify for the group. You can use smart groups to show the list of recently used instruments or instruments whose name or description contains a specific keyword.

Creating a Static Group

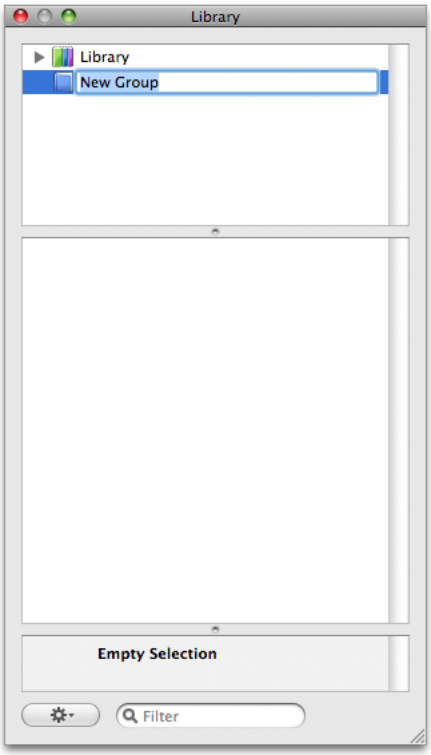
To create a static group, choose New Group from the Library window's action menu, as shown in Figure 2-4.

Figure 2-4 Creating a group



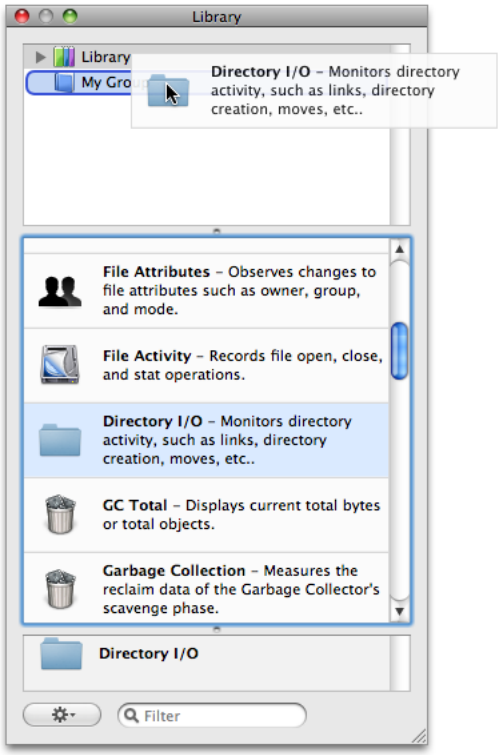
Instruments creates a new group and adds it to the Library window. To see the new group and edit its name and contents, drag the separator bar beneath the pop-up menu to open an upper pane showing the group structure of the Library, as shown in Figure 2-5. Click on the group and give it a new name.

Figure 2-5 Editing the name of a group



To add an instrument to a static group, click Library to display the list of instruments, select the instrument, and drag it to the group in the upper pane as illustrated in Figure 2-6.

Figure 2-6 Adding an instrument to a custom group



You can create hierarchical organizations of instruments using static groups. Static groups can include other static groups and they can also include smart groups. Smart groups cannot contain other groups, however. To create a hierarchical set of groups, you can do one of the following:

- Drag an existing group to the parent static group.
- Select a static group and choose New Group or New Smart Group to create a new item as a child of the selected group.

A group that contains other groups displays its own instruments plus the instruments contained in all of its child groups. If the same instrument is present in multiple groups and the Show Group Banners option is enabled, then the same instrument is listed under each group that contains it. If the Show Group Banners option is disabled, however, the Library window coalesces identical copies of an instrument into a single entry.

To remove an instrument from a static group, do the following:

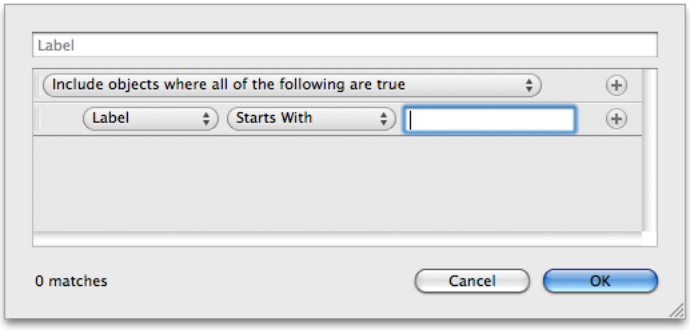
1. Select the group.
2. In the group, select the instrument you want to remove.
3. Choose Remove From Group from the action menu or simply press the Delete key.

To remove a static 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. Removing a static group removes the group and any nested groups it contains. This operation does not, however, delete the instruments contained in the group from the library. You can always find the instruments again by looking in the Library group.

Creating a Smart Group

To create a new smart group, choose New Smart Group from the Library window's action menu. Instruments displays the rule editor shown in Figure 2-7. The Label field identifies the name of the group as it appears in the Library window. The rest of the sheet is dedicated to configuring the rules that determine which instruments reside in the group.

Figure 2-7 The smart group rule editor



Every smart group must have at least one rule. You can add additional rules, as needed, using the controls in the rule editor and configure the group to apply all or some of the rules. Table 2-1 lists the criteria you can use to match instruments.

Table 2-1 Smart group criteria

Criteria	Description
Label	Matches instruments based on their title. This criterion supports the Starts With, Ends With, and Contains comparison operators.
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.

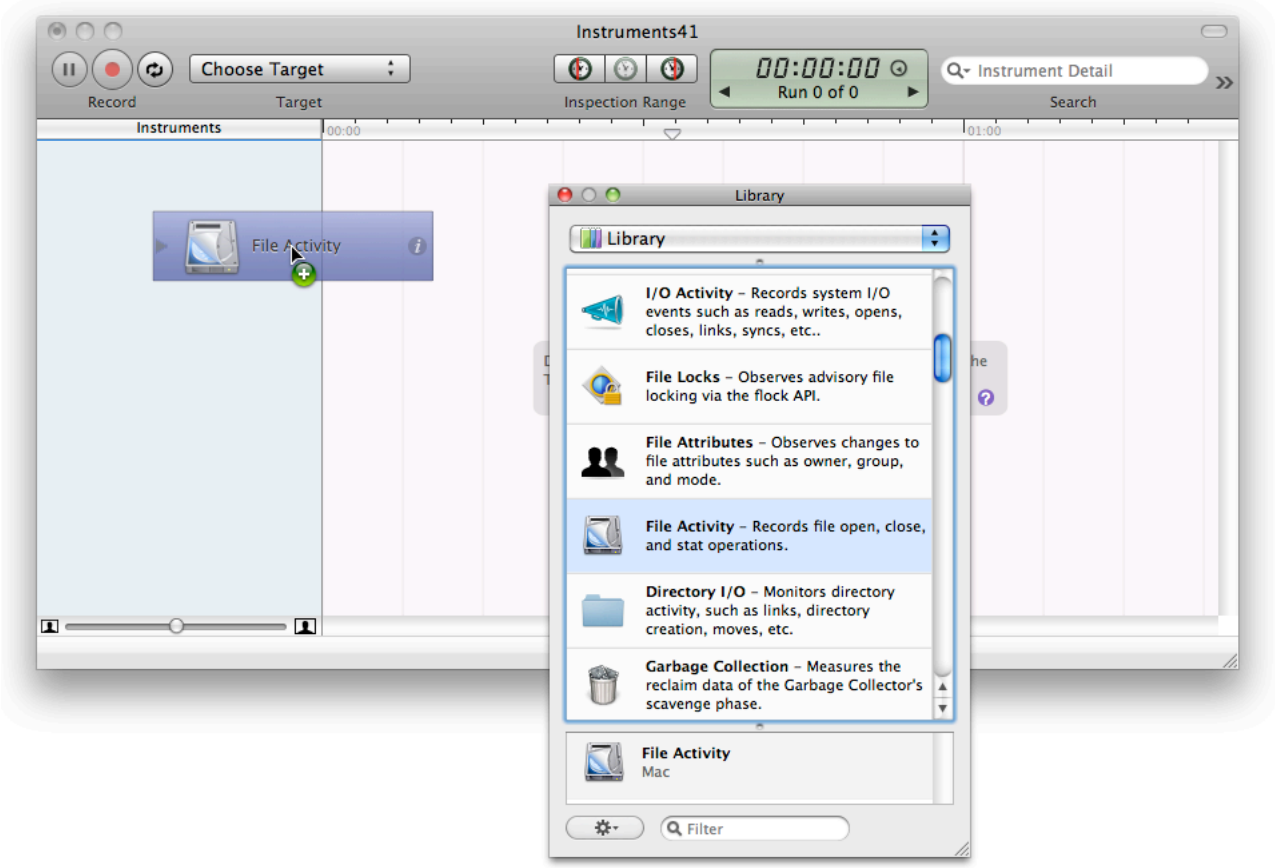
To edit an existing smart group, select the group in the Library window and choose Edit <*Group Name*> from the action menu, where <*Group Name*> is the name of your smart group. Instruments displays the rule editor again so that you can modify the existing rules.

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 and Removing Instruments

To add an instrument to a trace document, drag it from the Library window to the Instruments pane or Track pane of your trace document, as shown in Figure 2-8.

Figure 2-8 Adding an instrument



You can add as many instruments as you like to a trace document. Although some instruments can track all system processes, many track only a single process. For those instruments, you can use multiple instances of the instrument and assign each one to a different process if you like. In this way, you gather similar information for multiple programs simultaneously. For example, you might do this to sample a server process and one of its corresponding client processes and examine the overall interaction pattern.

To remove an instrument from a trace document, select the instrument in the Instruments pane and press Delete.

Configuring an Instrument

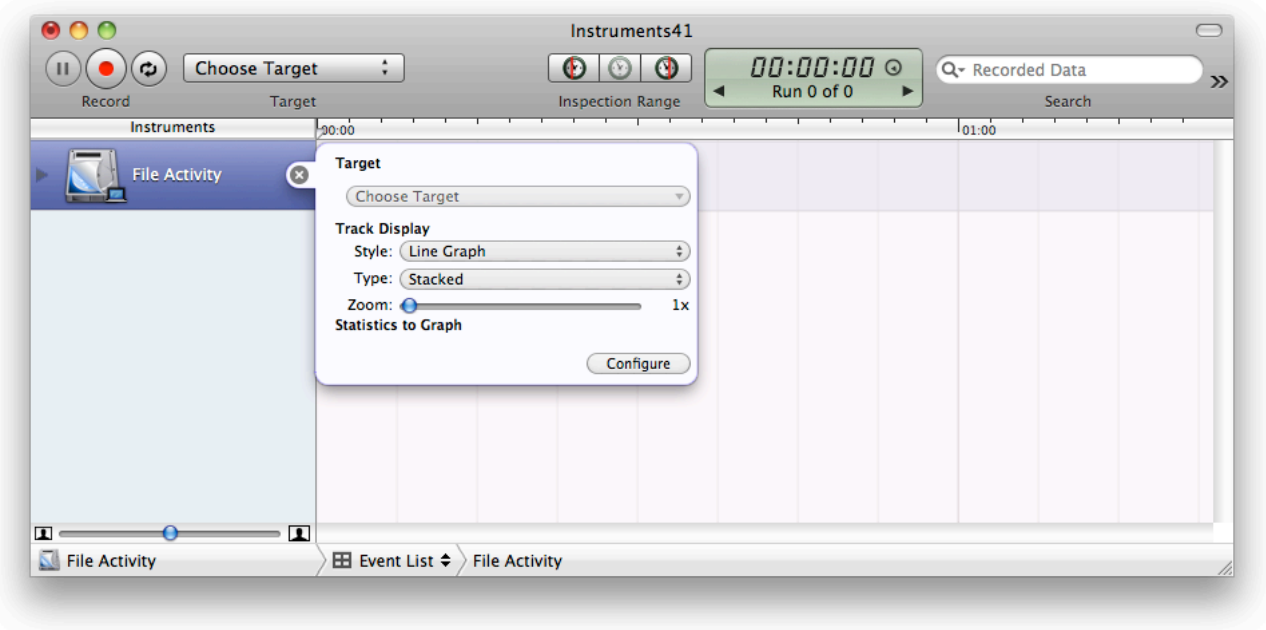
Most instruments are ready to use as soon as you add them to the Instruments pane. Some instruments can also be configured using the instrument inspector. The content of the inspector varies from instrument to instrument. Most instruments contain options for configuring the contents of the track pane and a few contain additional controls for determining what type of information is gathered by the instrument itself.

To open the inspector for a given instrument, select the instrument and do one of the following:

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

The inspector appears next to the instrument name, as shown in Figure 2-9. To dismiss the inspector, click the close button. (The commands you used to open the inspector can also be used to close it.)

Figure 2-9 Inspector for the File Activity instrument



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.

The Zoom control found in most inspectors controls the vertical magnification of the trace data in the track pane. Changing the zoom value changes the height of the instrument in the track pane. The View > Decrease Deck Size and View > Increase Deck Size commands do a similar job of decreasing and increasing the track pane magnification for the selected instrument.

For a list of instruments, including their configurable options, see "Built-in Instruments" (page 77).

Recording Trace Data

After you decide which instruments you want to use to gather data, the next step is to choose the process you want to trace and begin recording data. How you select a process is dependent on the instruments in your trace document. Some instruments allow tracing of all system processes, others require you to record data for a single process. Some instruments even expect you to launch the process from Instruments so that they can gather data from the beginning of its execution.

Instruments provides several options for initiating a trace that make it easier to integrate Instruments into your development cycle. In this chapter, you learn how to select a process for your trace document and begin recording data using the available options in Instruments.

Choosing Which Process to Trace

Before you can start collecting data, you must tell Instruments which process you want to trace. You do this by designating a target process (or processes) using the Target menu in the Instruments toolbar. This menu provides the following options:

- **All Processes**. Configures your document to trace all processes on the system.
- **Attach to Process**. Configures your document to trace an already running process.
- **Choose Target**. Configures your document to launch and trace a process. (If the process is already running, Instruments launches a new copy and traces it.)
- **Instrument Specific**. Select this item to assign different trace targets to individual instruments.

The following sections describe each of these options in more detail.

Tracing All Processes

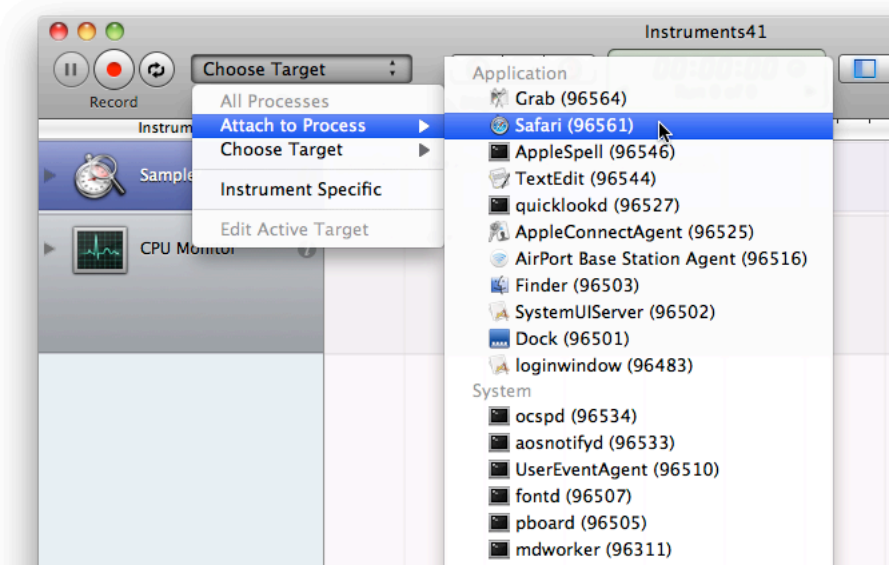
Some instruments are capable of collecting data for all processes currently running on the computer. You can use this ability to profile a type of event or activity across the entire system. For example, you can use the Disk Activity instrument to trace all read and write operations that occur on your computer over a particular time period. To trace all running processes, choose All Processes from the Target menu.

Note: The All Processes item is available only if all instruments in the Instruments pane support it.

Tracing an Existing Process

If the process you want to trace is already running on the computer, most instruments can attach to that process and start collecting data. To trace an existing process, choose Attach to Process from the Target menu and select the process to trace, as shown in Figure 3-1.

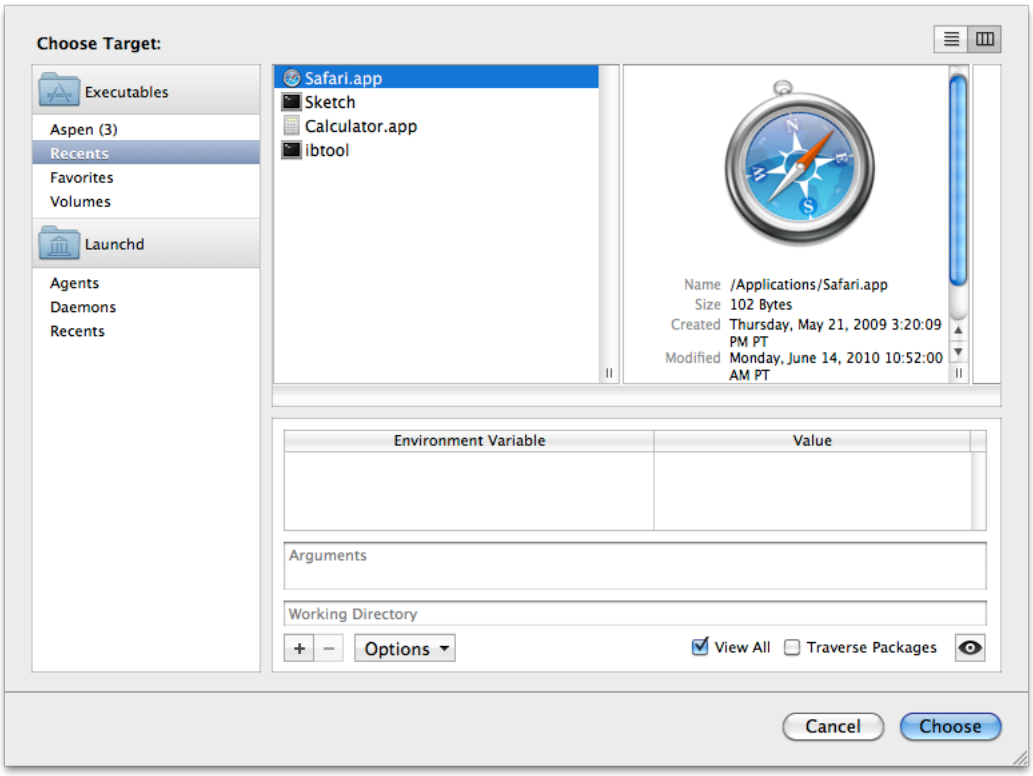
Figure 3-1 Tracing an existing process



Tracing a New Process

If the process that you want to trace is not already running on the computer, or if you want to control the conditions under which that process is launched, choose Choose Target from the Target menu. Instruments remembers any recently launched processes and adds them to the Choose Target submenu for quick access. If the process you want to launch is not in that menu, choose Choose Target to display the dialog shown in Figure 3-2.

Figure 3-2 Choosing a target to launch



The Choose Target dialog lets you select the program to launch and also lets you specify how to launch the selected program. Table 3-1 lists the additional controls in this dialog and explains how you use them.

Table 3-1 Options for launching an executable

Control	Description
Environment Variable	Identifies environment variables you want to set before running the process. You might use this option if your program has debugging options that are enabled using an environment variable. Use the plus (+) and minus (-) buttons to add or remove environment variables.
Arguments	Use this field to specify any launch arguments for the application. The arguments you specify are the same ones you would use from the command line when launching the application there.
Options	Use this menu to specify other runtime options. For example, you can direct the application's output to the Instruments console or the system console, or discard the output. You can also specify whether the application is launched in 32-bit or 64-bit mode.
View All	Sets the specified application as the target for all instruments in the trace document. This option is enabled by default. Disabling it lets you assign different targets to different instruments in your trace document. You might use this feature when you have two copies of the same instrument or when you want to trace the behavior of two processes running side by side.

Control	Description
Traverse Packages	Displays bundles (such as applications and plug-ins) as a navigable directory structure. Use this feature if the executable you want to run is inside a bundle.

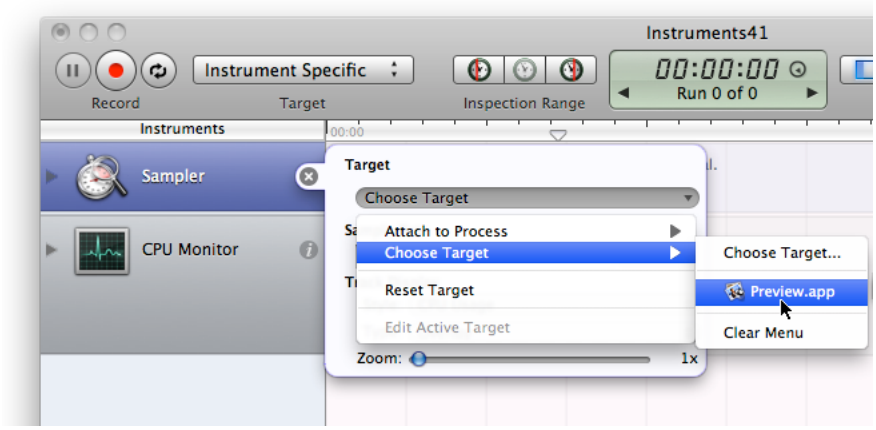
Specifying a Different Target for Each Instrument

When you choose All Processes, Attach to Process, or Choose Target from the Target menu, Instruments sets the default target for all instruments to the program you choose. There are occasions, however, when you might want to set a different target for each instrument. You might want to sample two different programs at the same time, for example, using the Sampler instrument. If you have some instruments that are capable of tracing all processes, you might want them to do just that while single-process instruments sample just one process.

To assign trace targets on a per-instrument basis, do the following:

1. In the Target menu, select the Instrument Specific option.
2. Select one of your instruments.
3. Open the instrument inspector by clicking its information icon.
4. In the inspector window, use the pop-up menu in the Target section to set the target for the instrument, as shown in Figure 3-3.

Figure 3-3 Choosing per-instrument trace targets



5. Repeat steps 2 through 4 for each of your other instruments.

Collecting Data

After you've selected a process to trace, you are ready to start collecting data. You can collect data in one of two different recording modes:

- **Immediate display**. During the measurement interval, Instruments displays the collected data in the Track and Detail panes immediately. The Time control in the Instruments toolbar also shows you how much time has elapsed since you began recording. In this mode, Instruments impacts system performance because the UI is being updated continuously. This mode is the default in Instruments. To select immediate display, choose File > Record Options > Immediate Display.
- **Deferred display**. Instruments defers the display of the collected data until the recording stops. During the measurement interval, Instrument's impact on the system is minimal. When you stop recording, Instruments performs postprocessing and displays the data. Deferred display is an important feature if the application being measured is highly performance-sensitive. To select deferred display, choose File > Record Options > Deferred Display.

The recording mode persists with the document.

Click the Record button (or choose File > Record Trace) to begin gathering the trace data. When you click Record, Instruments launches the specified executable or attaches to the specified processes and starts gathering data. To stop collecting data, click the Stop button or choose File > Stop Trace.

Note: When you click the Record button, Instruments may display one or more authentication dialogs. Many instruments require you to authenticate as an administrative user before they record any data. Instruments is a powerful tool that enables you to see into running applications, and as such, it should be used only by authorized users.

During recording, if you want your program to continue running, but do not want Instruments to gather data, press the Pause button in your trace document. Instruments stops gathering data temporarily but does not stop the current recording run. Pressing the Resume button causes Instruments to continue gathering data at the current recording time. Thus, pausing and resuming leaves a data gap in the Track pane.

Launching Instruments Using Quick Start Keys

Quick Start keys are global key combinations that let you launch the Instruments application and begin collecting trace data immediately using a specific document template. You can use this feature if you are running an application and see something—such as a bug or unresponsive behavior—that you want to capture right away. You can associate different key combinations with different Instruments templates to capture different types of behavior.

To assign a Quick Start key combination, open the Instruments preferences and navigate to the Quick Start pane. This pane displays the list of Instruments templates that you can assign to key combinations. All templates start without a key combination. To assign one, find the row containing the desired template and double-click its Key column to create an editable cell. While the cell is in edit mode, press the key combination you want to use. For example, to assign the key combination Command-Option-1, press the Command, Option, and number 1 key all at the same time.

To remove a key combination from a template, select a row and press Delete.

Note: Quick Start key combinations should use at least two key modifiers (Command, Option, Control, Shift). You should avoid choosing key combinations that are commonly used by other applications.

To gather data for an application using a Quick Start Key, do the following:

1. Place the cursor over a window belonging to the application you want to trace.
2. Press the appropriate key combination to begin tracing.
3. Exercise the application.
4. When you want to stop tracing, you have two options:
   - Place the cursor over one of the application's windows and press the key combination again.
   - Find the trace document that was opened and press the Stop button.

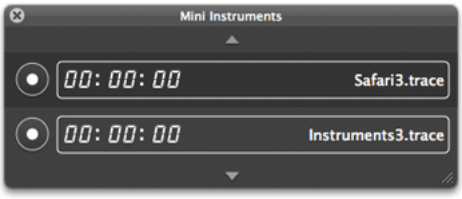
Because Quick Start Keys require you to move the cursor over one of the application's windows, you can initiate multiple trace sessions for different applications using the same Quick Start Key without stopping any of the previous traces. You could also start different types of traces for the same application using different key combinations and have them all gathering data at the same time.

Running in Mini Mode

Mini mode provides a way for you to minimize the visual footprint of the Instruments application while gathering your data. When you are gathering data from certain types of applications, particularly graphics-oriented applications, there may be times when you want to gather data while remaining focused on your own application. Mini mode hides all open trace documents and in their place displays a small floating window that you can use to start and stop tracing. An added advantage of mini mode is that the Instruments application itself requires less drawing and therefore has less of an impact on system performance.

Figure 3-4 shows the Mini Instruments window with several trace documents open. Instruments shows only three trace documents at a time in mini mode, but you can use the up and down arrows to find the document you want.

Figure 3-4 The Mini Instruments window



Clicking the record button next to a trace document begins recording data for that document. The document must already have its instrument and target process configured before tracing can begin. During tracing, Instruments displays the elapsed time since tracing began but displays no other instruments, controls, or data.

To enable mini mode, choose View > Mini Instruments. To disable mini mode, click the close box in the Mini Instruments window. You can also toggle between standard and mini modes by choosing View > Mini Instruments.

Running Instruments from Xcode

During development, you can launch your programs in Instruments directly from the Xcode 3 user interface. This integration lets you quickly launch Instruments and gather trace data in much the same way that you would launch your program and debug it using GDB.

The Run > Run with Performance Tool submenu in Xcode 3 provides several options for launching your program using the available performance tools, including Instruments. When launching your program in Instruments, you tell Xcode which Instruments template you want to use by selecting the appropriate menu item. Xcode launches Instruments, creates a trace document using the specified template, sets the target to your program, and tells Instruments to launch your program and start recording.

In addition to the existing templates in the Run with Performance Tool submenu, Xcode lets you add custom trace templates to this menu as well. For information on how to save a trace template and add it to this menu, see "Saving an Instruments Trace Template" (page 63).

Connecting Wirelessly to an iOS Device

To collect data from an iOS application that uses an external accessory, you can create a wireless connection between Instruments and your device. You may also use a wireless connection when a wired connection is impractical or inconvenient.

Software requirements: This feature is available in iOS SDK 3.1 and later.

In general, if you do not need to attach an external accessory to your device, you should use a wired connection. The wireless connection is slower than the wired connection and requires more power from the device to accommodate the heavy traffic that can come from Instruments. As a result, using a wireless connection drains the device's battery more rapidly than using a wired connection.

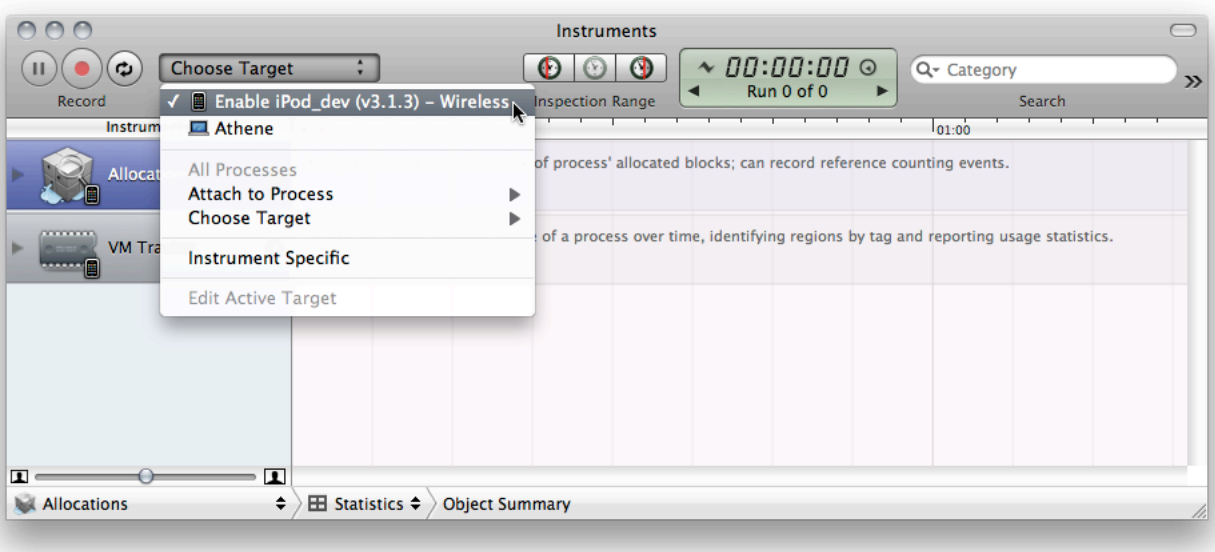
Important: For Instruments to be able to connect wirelessly to your device, both your Mac and the device must have Wi-Fi networking turned on and must be connected to the same Wi-Fi access point. This access point must be configured to provide Bonjour services.

To create a wireless connection between Instruments and your device:

1. Connect your device to your computer using a USB cable.
2. In Xcode, install your application on the device.
3. In Instruments, open or create a trace document.

4. Press the Option key and choose "Enable <device_name> - Wireless" from the Target menu, as shown in Figure 3-5.

Figure 3-5 Creating a wireless connection between Instruments and an iOS device



After a moment, a wireless target appears in the Target menu, named "<device_name> Wireless." If the wireless target does not appear, your Wi-Fi access point may not be configured to support Bonjour traffic.

At this point you have two Instruments servers running on your device: one using the wired connection and the other using the wireless connection.

5. Disconnect the USB cable from your device and connect the external accessory.

Even if you are not using an external accessory, you should disconnect the USB cable from your device to conserve memory.

Now you can use Instruments to gather performance data about your application, as described in "Choosing Which Process to Trace" (page 31).

The wireless connection remains in effect even after quitting Instruments or locking your device. You can continue using the wireless connection while the network connectivity conditions described earlier remain.

To terminate the wireless connection:

1. In Instruments, open a trace document.
2. Press the Option key and choose "Disable <device_name> - Wireless" from the Target menu.

After a moment, the "<device_name> Wireless" target dims, indicating the wireless connection and Instruments server are terminated.

Working with a User Interface Track

A user interface track records a series of events or operations in a running program. After the events are recorded, you can replay that track multiple times to reproduce the same sequence of events over and over. Each time you replay a user interface track, you can collect data using other instruments in your trace document. The benefit of doing so is that you can then compare the data you gather on each successful run and use it to measure the changes in your application's performance or behavior.

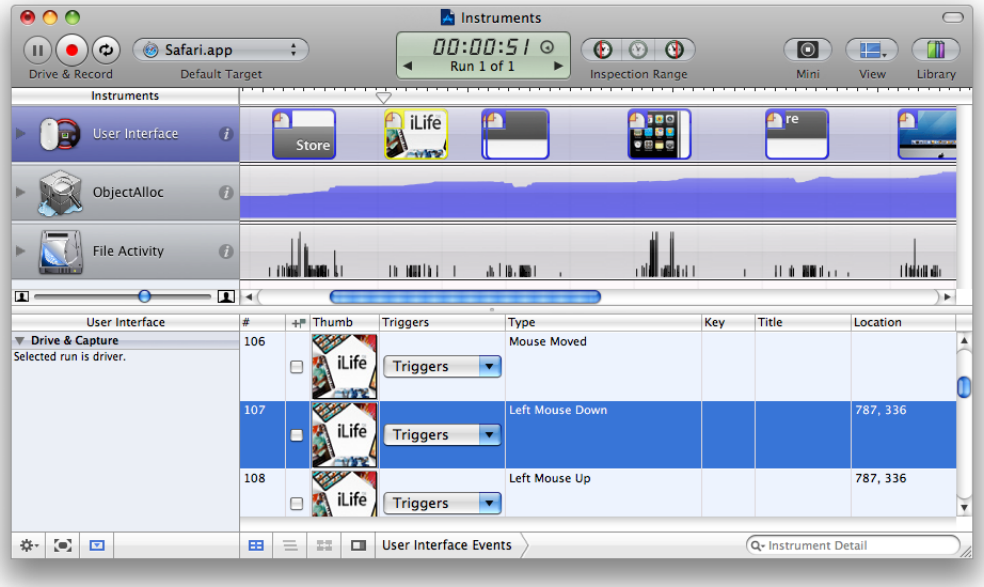
Recording a User Interface Track

You record a user interface track with the User Interface instrument. You add this instrument to your trace document just as you would any other instrument. When you click the Record button in your document, the User Interface instrument gathers events related to user input, such as mouse and keyboard events. It collects these events and displays them in the track pane for you to see and examine.

Note: If you create a new trace document using the UI Recorder template, Instruments automatically adds the User Interface instrument to the document for you.

After you have gathered a set of events for the user interface track, the title of the Record button changes to Drive & Record. Pushing this button again causes Instruments to drive the user interface, replaying the sequence of events you recorded previously. While it does this, the other instruments in your trace document gather data as usual. Upon completion of the event stream, you should have a new run of data to examine. Figure 4-1 shows a trace document with a User Interface track and several other instruments. Here, the user interface track already contains a sequence of events that are ready to be replayed.

Figure 4-1 Recording a user interface track



Important: User interface recording takes advantage of accessibility features normally used by screen readers and other assistive devices for disabled persons. To ensure that your computer supports this feature, open the Universal Access pane of System Preferences and make sure the "Enable access for assistive devices" setting is enabled.

The Detail pane for the User Interface instrument lists the events that were recorded, the location of the event, and the key that was pressed, if any. The instrument can also capture a screen shot of the area affected by the user action. A thumbnail version of the screen shot is shown in both the track pane and the detail pane. Each event is color-coded according to its type:

- Mouse events are blue.
- Key events are green.
- System events are yellow.

Rerecording a User Interface Track

If, after capturing a sequence of events, you decide that you did not get the sequence right, you can go back and recapture the sequence as needed until you get it right. Before you recapture the sequence, though, you must tell Instruments not to drive the user interface using the old event sequence. The Action section of the User Interface inspector contains a pop-up menu that lets you specify how you want that instrument to run. Normally, after you capture a sequence of events, the Action setting is automatically set to Drive. To

recapture, you must change the setting to Capture. After you have done that, you can begin recording the new sequence. Because the Action setting switches back to Drive after each recording, recapturing a sequence several times requires that you change the Action setting back to Capture every time you record.

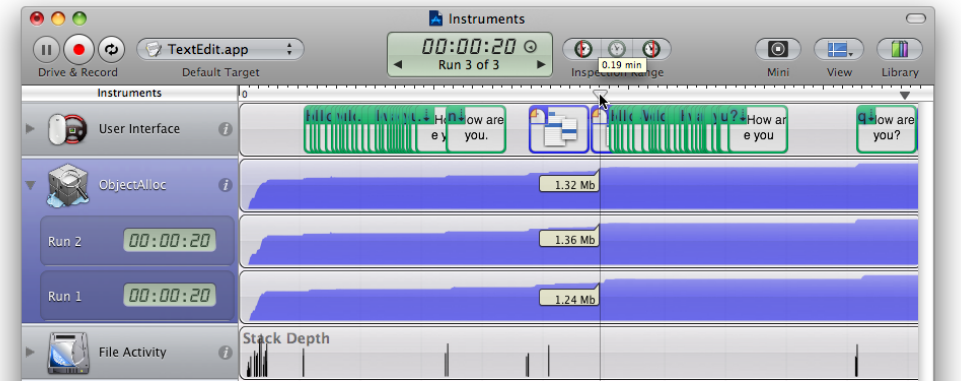
Replaying a User Interface Track

After you have recorded a user interface track, you can replay that track to reproduce the sequence of user events that you recorded. After recording a sequence of events, Instruments automatically changes the title of the Record button to Drive & Record. Clicking this button launches the selected application and performs each of the actions recorded in the user interface track. At the same time, Instruments begins collecting data for any active instruments in the Instruments pane.

After each successful run of the user interface track, Instruments displays the results of that run in the track pane. To change the active set of results in the track pane, you can use the arrows in the Time control. Clicking these arrows cycles through each of the available data sets, updating the display for each of the available instruments.

You can also view all of the runs for an instrument at the same time so that you can compare them. Each instrument includes a disclosure triangle on its left side. Clicking this control expands the instrument and displays the data from all of the previous runs side by side, as shown in Figure 4-2.

Figure 4-2 Viewing multiple runs



While collecting trace data, you can have Instruments repeat the events in the user interface track multiple times in succession by clicking the Loop button in the document toolbar. The state of the button is sticky, so clicking it enables looping and clicking it again disables it. When looping is enabled, Instruments runs the event sequence in a continuous loop, gathering a new set of trace data each time through the loop. You might use this feature to repeat a sequence of events that exhibits undesirable behavior after a number of tries. You can also use looping to stress-test your application by keeping it constantly busy.

Viewing and Analyzing Trace Data

Because Instruments lets you gather data from multiple instruments simultaneously, the amount of data available to you can quickly get overwhelming. Fortunately, Instruments devotes much of its interface to organizing and displaying data in ways that help you analyze and navigate that data. Understanding the different areas of your trace document window is essential to spotting trends and finding potential problem areas.

Every trace document includes the following interface elements:

- The Track pane
- The Detail pane
- The Extended Detail pane
- The Run Browser

Each of these elements show you the exact same data in your trace document; they just show it in very different ways, ranging from high-level overviews to detailed information about each data point. The idea is to give you different ways to visualize your data. You can view the high-level data to look for general trends, and then look at the detailed data to find out exactly what is happening in your code and formulate ideas on how to fix potential issues.

In addition to the user interface options, many instruments can be configured to present their data in several different ways. You can use these configuration options during the analysis phase to get fresh perspectives on the gathered data.

In this chapter, you learn how to use the information provided to you by Instruments and how you can modify the way that information is presented it to suit your workflow needs.

Tools for Viewing Data

The following sections describe key elements of the Instruments user interface and how you use those elements to view trace data.

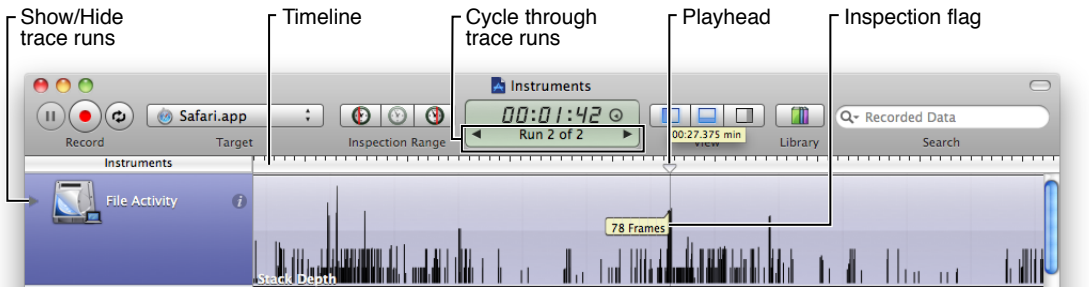
The Track Pane

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 program. For example, a spike in a memory usage graph indicates a place where your program 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 application becomes unresponsive. If the graph for the Spin Monitor is relatively empty, you know that your application 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 where you want to investigate. Clicking in the timeline moves the playhead to that location and displays a set of inspection flags, which 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 "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 program.

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

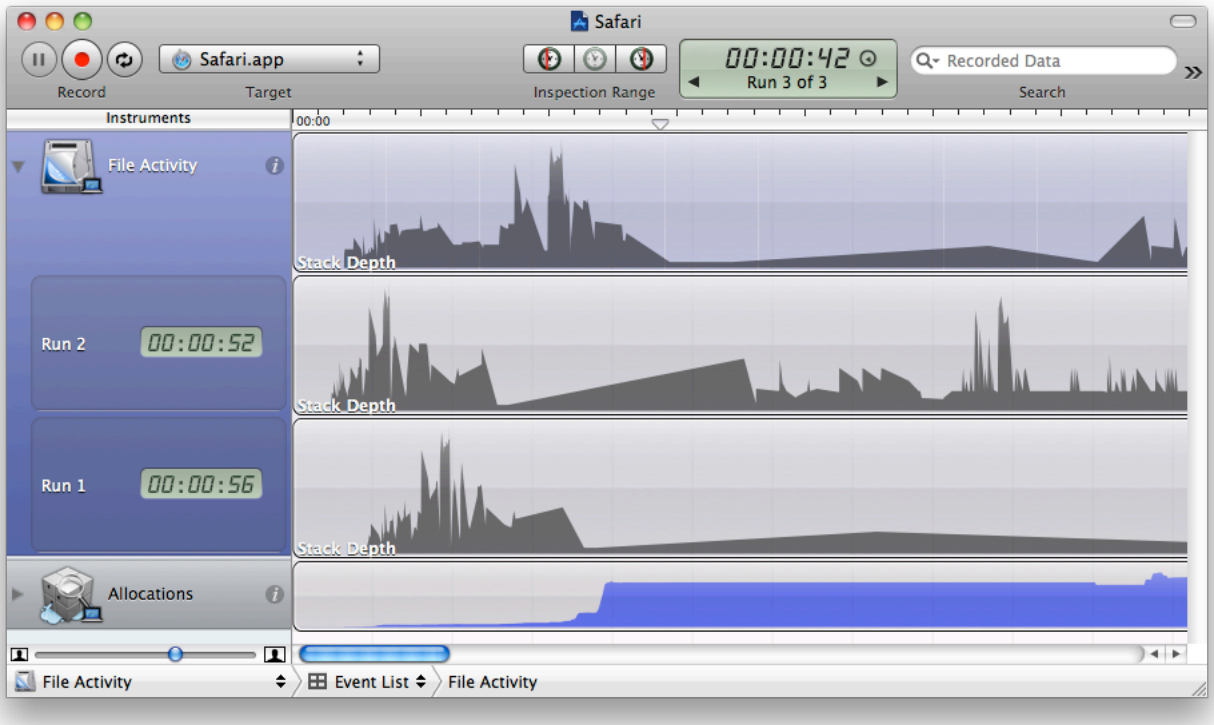
Viewing Trace Runs

Each time you click the Record button in a trace document, Instruments starts gathering data for the target processes. Rather than append the new data to any existing data, Instruments creates a new trace run to store that data. A **trace run** constitutes all of the data gathered between the time you clicked the Record button and stopped gathering data by clicking the Stop button. By default, Instruments displays only the most recent trace run in the track pane, but you can view data from previous trace runs in one of two ways:

- Use the Time/Run control in the toolbar to select which trace run you want to view.
- Click the disclosure triangle next to an instrument to display the data for all trace runs for that instrument.

Figure 5-2 shows an instrument that has been expanded to show multiple trace runs. In this example, several trace runs have been collected, each of which was generated from a slightly different set of events. You might gather multiple trace runs when you are trying to reproduce a problem that does not occur every time, creating new trace runs until the problem surfaces. If you want to compare multiple trace runs using the exact same set of events, however, you need to create a user interface track, a process that is described in "Working with a User Interface Track" (page 39).

Figure 5-2 Viewing multiple runs of an instrument

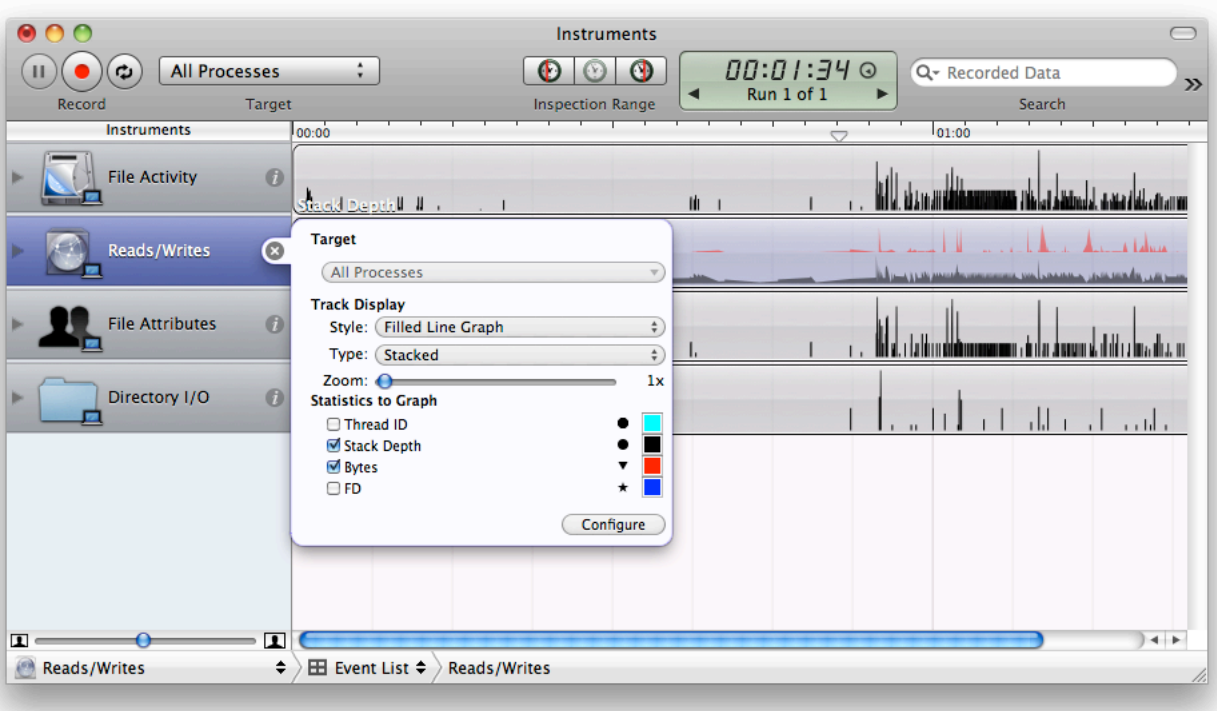


When you save a trace document, Instruments normally discards the data from all runs except the one that is currently selected. It does this to keep the size of your trace documents from becoming too large. If you want to save all trace runs for your documents, go to the General tab of the Instruments preferences and uncheck the Save Current Run Only option.

Displaying Multiple Data Sets for an Instrument

Some instruments are capable of displaying multiple streams of data inside their corresponding track pane. You can configure which streams of data are displayed using the instrument inspector, as shown in Figure 5-3. The Statistics to Graph section lists all of the integer-based data values gathered by the instrument. Selecting the checkboxes in that section adds the corresponding statistics to the track pane. To edit the list of statistics in the Statistics to Graph section, click the Configure button.

Figure 5-3 Configuring the statistics to graph

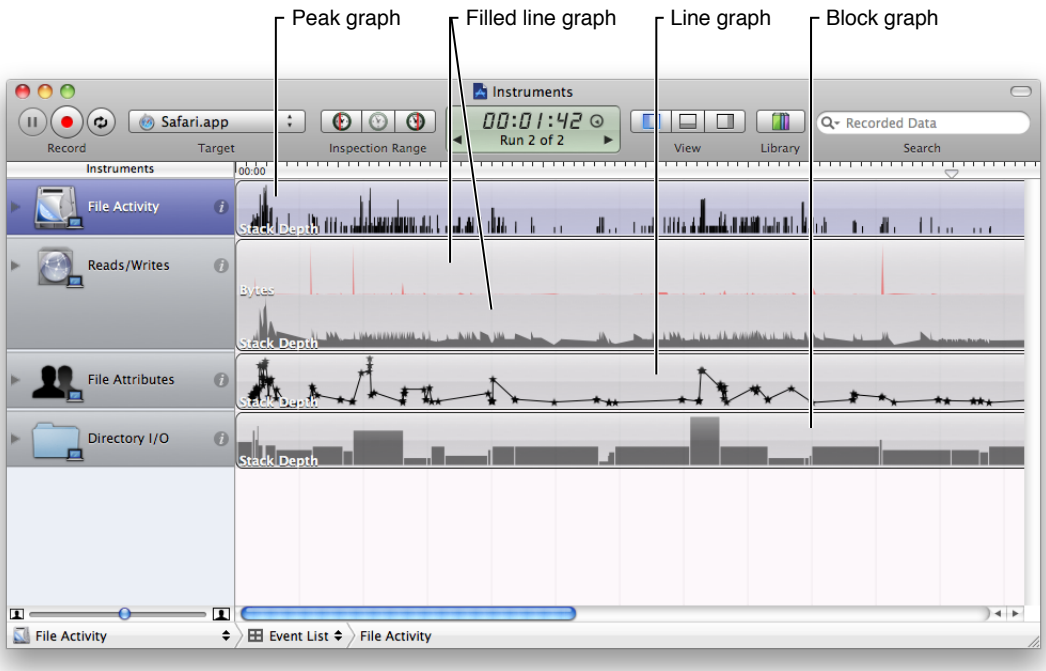


For information about the data gathered by each instrument, see "Built-in Instruments" (page 77).

Changing the Display Style of a Track

Instruments provides several different styles for graphing data in the track pane. Most instruments choose a style that makes the most sense for the type of data they are displaying. You can change the selected style for an instrument using the Style pop-up menu in the instrument's inspector. Figure 5-4 shows several of the styles.

Figure 5-4 Track styles



Note: The style option you choose affects all of the data streams displayed by the instrument.

Zooming in the Track Pane

Instruments supports zooming in and out on track pane data along both the vertical and the horizontal axes.

- To change the horizontal (time) scale, use the slider control below the instruments pane.
- To change the vertical (amplitude or volume) scale, select an instrument and do one of the following:
  - Open the instrument's inspector and use the Zoom slider there.
  - Select the View > Increase Deck Size menu item to increase the zoom factor.
  - Select the View > Decrease Deck Size menu item to decrease the zoom factor.

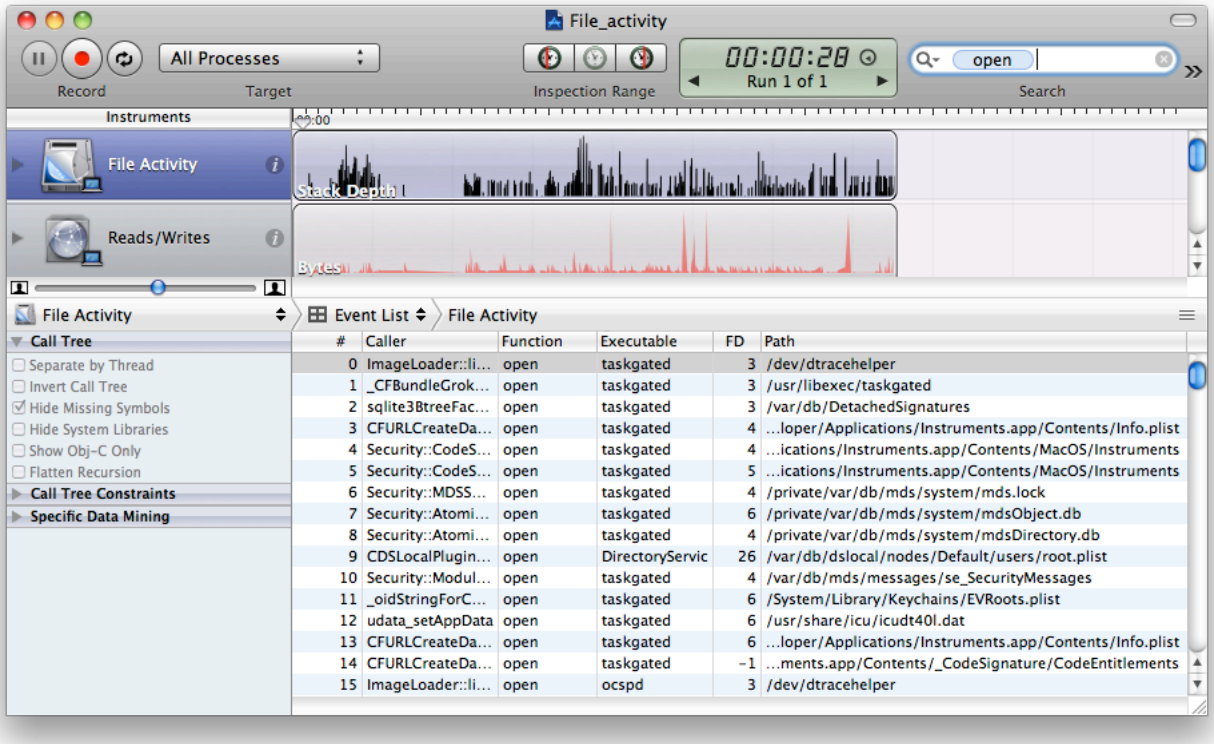
For horizontal zooming, Instruments expands or contracts the track pane around the current playhead position. If you set the playhead to a particular point before zooming, you can zoom in or out on the data under that point.

The Detail Pane

After you identify a potential problem area in the track pane, you 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 5-5 shows the Detail pane associated with the File Activity instrument, which records information related to specific 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 "Built-in Instruments" (page 77).

Figure 5-5 The Detail pane



To open or close the Detail pane, do one of the following:

- Choose View > Detail.
- Click the Detail 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. The Detail pane supports the following formatting modes:

- **Table mode** displays a flat list of aggregated sample data.
- **Outline mode** displays a hierarchical list of samples, typically organized using stack trace or process information.
- **Diagram mode** displays a list of the individual samples.

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. For example, the Sampler instrument does not support the diagram mode, but the Activity Monitor instrument supports all three modes.

When displaying data in Outline mode, 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

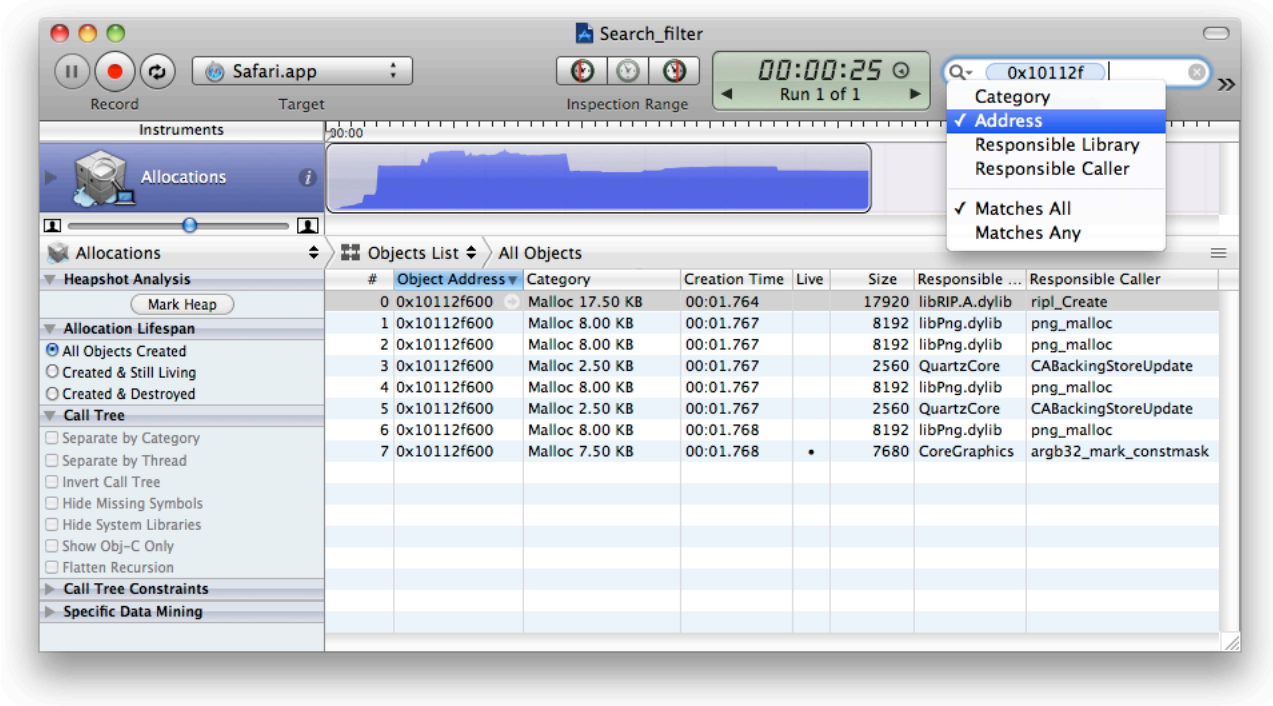
You can sort the information displayed in the Detail pane according to the data in a particular column. To do so, click the appropriate column header. The columns in the Detail pane differ with each instrument.

Searching in the Detail Pane

You can type a string in the search field in the toolbar to narrow down the list of information displayed in the Detail pane. By default, Instruments applies the specified search string against all of the data recorded by the instrument. With some instruments, however, you can refine the scope of your search even further. For example, use the Allocations instrument to apply the search string against specific subsets of the instrument data, such as the library or routine that created the memory block. You can even search for objects at a specific memory address.

The filter scoping criteria varies from instrument to instrument. To specify the desired scope, click the magnifying glass in the search field and choose from the available options. Figure 5-6 shows the search options for the Allocations instrument.

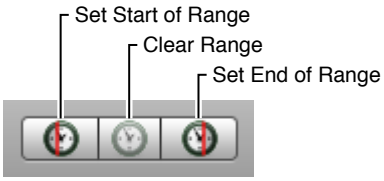
Figure 5-6 Filtering the Detail pane



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 5-7) to focus on the data collected in a specific time range.

Figure 5-7 Inspection Range control



To mark a time range for inspection, do the following:

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 clicking and dragging in the track pane of the desired instrument. Clicking and 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 button in the center of the Inspection Range control.

The Extended Detail Pane

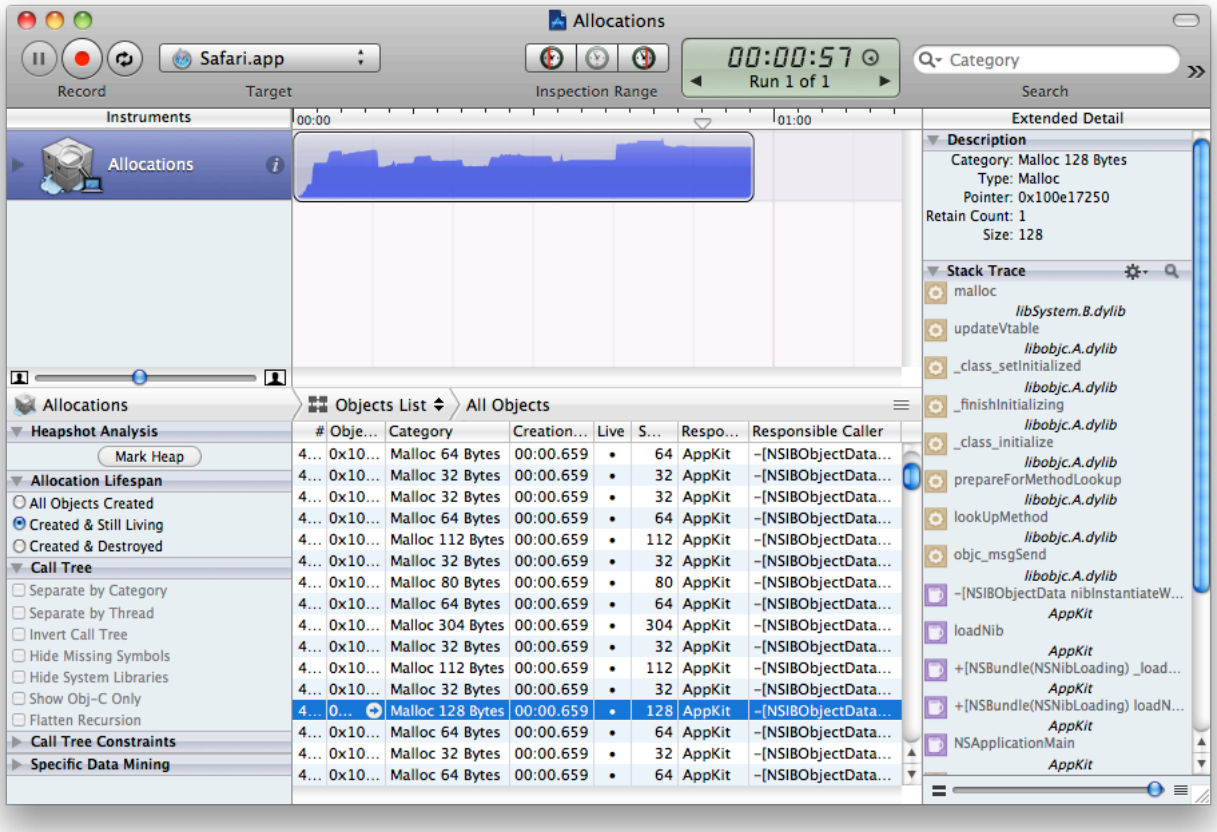
For some instruments, the Extended Detail pane shows additional information about the item currently selected in the Detail pane. To open or close the Extended Detail pane, do one of the following:

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

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. For information about what a particular instrument displays in this pane, see the instrument descriptions in "Built-in Instruments" (page 77).

Figure 5-8 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 5-8 Extended Detail pane



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.

Action	Description
Copy Selected Frames	Copies the stack trace information for the selected frames to the pasteboard so that you can paste it into other applications.

If you have an Xcode project with the source code for the symbols listed in a stack trace, you can double-click a symbol name in the Extended Detail pane to display the relevant source code. The source code is annotated with performance information.

Stack traces are collapsible using the slider control at the bottom of the Extended Detail pane. This is called backtrace compression with filtering. The purpose of this feature is to reduce the detail in the stack trace and display only what matters.

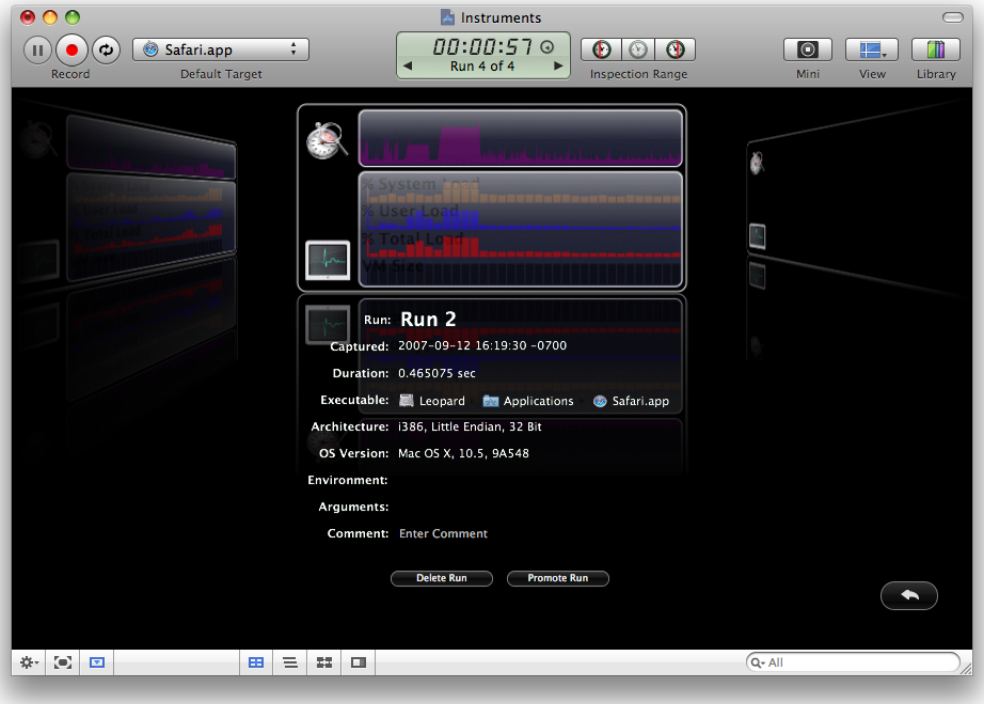
The Run Browser

The Run Browser is a way to view and manage previous runs quickly. If your trace document contains numerous runs, you can use this mode to scan those runs for the one you want and promote it to the top of the list. You can also delete runs and add comments to runs from this view.

To open the Run Browser, select View > Run Browser.

"The Run Browser" shows the Run Browser view in Instruments. Clicking views on either side of the selected view scrolls the new view into focus. To enter a comment for a view, double click the text in the Comment field to make it editable. To exit the Run Browser view, click the return arrow button in the bottom-right portion of the window.

Figure 5-9 The Run Browser



Analysis Techniques

Gathering and viewing trace data is easy. Analyzing that data and locating potential problems is hard. It is this latter task where the art of performance tuning and debugging comes into play. Recognizing patterns in the massive amounts of data gathered by performance tools can be daunting, even for experienced developers. This is where having the right tools (and knowing how to use them) makes all the difference. The instruments that come with the Instruments application provide many different options for organizing and filtering trace data. The following sections describe some of the behaviors and options available for specific instruments and how you use those instruments to identify issues in your code.

Analyzing Data with the Sampler Instrument

The Sampler instrument is a tool for performing a statistical analysis on a running application. Performing a statistical analysis involves stopping an application at regular intervals and recording information about what was executing at that moment in time. For each thread, the Sampler instrument records information about the functions and methods currently on the stack, including the name of the function or method and the module that owns it. After gathering the data, it coalesces the call stack information from the individual samples to form a master call tree for the application. This tree shows all of the execution paths that were seen during sampling and how many times each one was seen.

The advantage of statistical sampling is that it is a lightweight and convenient way to find out what your application is doing over a period of time. The technique can be used on any application without specially instrumenting the code, and it usually offers a reasonably accurate picture of your application's runtime behavior.

The disadvantage of statistical sampling, however, is that it does not give you a 100% accurate picture of what your application is doing. Because it only takes snapshots of the call stack at periodic intervals, the Sampler instrument does not record an exact history of all of the functions and methods that were executed. With typical sampling intervals on the order of 1 to 10 milliseconds, it is possible for a lot of functions and methods to be called between samples. Despite this seemingly inaccurate picture, statistical sampling does work for most applications when enough samples are gathered. Over time, the distortions caused by the sampling interval tend to smooth out as more samples are gathered. As a result, statistical sampling is still a good way to gather information about your application quickly and without too much effort.

Note: The Sampler instrument replaces the Sampler application, which is not available in Mac OS X v10.5 and later.

Analyzing the Call Stack Data

The purpose of the Sampler instrument is to show you where your application is spending its time. It does this by showing you which functions were called and how often they were called.

The place to start in the Sampler tool is the Track pane. By default, the track pane graphs the maximum call stack depth for all the threads of the application. (You can also change this graph to show the CPU load instead.) Because it offers a visual approach, the Track pane can help you spot trends in your code. Repeated patterns can indicate similar code paths being executed. The shape of different sections can also give some indication as to what they were doing. By clicking in an interesting area, you can then begin analyzing the data at that area using the Detail pane.

The Detail pane is where you do a more in-depth analysis of your code. The Detail pane supports both the Table and Outline viewing modes. Table mode presents you with the time-ordered list of samples. You can use this list to see what code was executing at a given point in your application. In Outline mode, you are presented with the sample data organized by call stack. From this view, you can look at each execution branch of each thread and see which ones have unusually large numbers of samples. Expanding each thread lets you see the individual methods and functions and the number of samples gathered for each one. To expand an entire hierarchy all at once, hold down the Option key while clicking the disclosure triangle.

In addition to starting at a thread's main routine and searching for heavy branches, you can also invert the call tree to start at the leaf nodes and see which methods and functions were called most often. Inverting the call tree can help quickly identify methods and functions that are perhaps being used too frequently. You can then expand the tree from these leaf nodes to find out who called that method or function and how often. While doing this, it is often helpful to display the number of milliseconds the given method or function spent running so that you can correlate the number of samples with the actual amount of time that function used.

Note: Although the Sampler instrument reports the amount of time spent in a given branch, these times are only approximations. You can use these values to determine roughly where your application spent its time but should not use them as performance metrics to determine how fast your code is.

The Extended Detail pane provides you with additional views of the sample data. In Table mode, the pane displays the data gathered by the selected sample, which includes the stack traces for all threads running when the sample was taken. In Outline mode, it shows the deepest stack trace containing the selected method or function.

Filtering the Contents of the Detail Pane

Table 5-2 lists the high-level configuration options you can apply to samples gathered with the Sampler instrument. You use these options to focus your analysis on the events that occurred at a particular time or that involve a particular portion of your code.

Table 5-2 Configuration options for the Sampler instrument

Configuration section	Description
Sample Perspective	Choose between displaying all samples that were captured or only those that were captured while the specified thread (or threads) were running. Viewing all samples can give you a complete picture of the behavior of a thread over a period of time, including how much time was spent blocked or waiting for data. Viewing only the running samples provides an approximate picture of how much time was spent actually executing your code. (The actual running time may differ somewhat from the time reported by Instruments so you should use the reported values only as a rough guide.)
Call Tree	Choose these options to flatten or hide uninteresting parts of the call tree. You can separate out symbols that were gathered from different threads of execution, hide missing symbols or libraries, flatten branches of the call tree that contain recursive calls, and more. These options help you trim irrelevant portions of the call tree and organize the remaining data in ways that make it easier to spot trends.
Call Tree Constraints	Choose the constraints for the data you want to view. You can use these configuration fields to prune the current data set. The Sampler instrument supports constraining data based on the number of samples gathered in a branch or the number of milliseconds spent executing a branch.
Active Thread	Choose the thread you want to analyze. Focusing on a specific thread displays only the samples for that thread, making it easier to see what the thread was doing. Viewing all threads lets you see all of the work being performed by your application.

The Call Tree configuration options offer several ways to trim down the call tree without removing any sample data. When you hide or flatten a set of symbols, Sampler applies the samples for those symbols to the calling function or method. This lets you remove any symbols that you cannot control and focus on your own code and how much time it took to execute.

Alternatively, the Call Tree Constraints actually remove sets of samples from the view to let you focus on code paths that meet specific criteria. For example, you might use a time-based constraint to focus on code paths that took at least 100 milliseconds to run.

When applying an instrument's configuration options, do not forget that you can also constrain the sample data based on when those samples were gathered. The Inspection Range control in each trace document lets you view the data from a specific set of sample points. This feature works in combination with all of the instrument's other configuration options. For more information on how to use the Inspection Range control, see "Viewing Data for a Range of Time" (page 50).

Analyzing Data with the Allocations Instrument

The Allocations instrument is a tool for tracking all of the memory allocations made by an application. You can then use that information to identify memory allocation patterns in your application and identify places where your application is using memory inefficiently. The Allocations instrument provides data trimming and pruning facilities that are equal to or better than the former ObjectAlloc application. Because it is integrated with the Instruments environment, you can also use the instrument to correlate your application's memory behavior with other types of behavior.

Because it tracks memory allocations over the life of an application, you must launch your application from Instruments so that the Allocations instrument can gather the data it needs. At launch time, the Allocations instrument uses existing hooks in the system to record information about allocation and deallocation events in your application, whether those events originated with the standard system malloc routines or your own custom malloc library. As data flows in, the instrument updates its displays in real time to show you how memory is being allocated.

The Allocations instrument works with applications that use the standard malloc functions (such as `malloc`, `calloc`, and `free`) but also works with garbage collected applications. In the latter case, the collector still issues `free` calls for GC-aware memory. The Allocations instrument also works with routines built on top of malloc, including the memory allocation routines in both Core Foundation and Cocoa.

Note: The Allocations instrument replaces the ObjectAlloc application, which is not available in Mac OS X v10.5 and later.

Analyzing the Object Allocation Data

The purpose of the Allocations instrument is to show you how your application is using memory. Memory is an important system resource that you should use wisely. Each memory allocation has both an immediate cost and a potential long-term cost. The immediate cost is the time it takes to allocate the memory, which could involve creating new virtual memory pages and mapping them into physical memory. It could also involve writing out stale memory pages to disk. Long term, keeping blocks resident in physical memory can trigger additional paging in the system, which as with all paging operations can seriously hamper performance.

Like all tools, the place to start in the Allocations tool is the Track pane. In its default configuration, the Track pane graphs the net amount of memory currently in use by your application. Using the instrument's inspector, you can change this view to show the allocation density, which shows you where the memory allocations occurred, or you can display the stack depth. The allocation density graph lets you see how frequently memory allocations occurred throughout your program. Spikes in the allocation density can indicate potential bottlenecks that you might want to mitigate by preallocating some blocks or being more lazy about other blocks.

Regardless of which display options you set using the inspector, the Track pane displays the allocations for all types of objects by default. To focus on a specific subset of memory allocations, you can use the Detail pane to configure which objects you want to include in the Track pane's graph. To focus on a particular object type or block size, open the Detail pane and set it to Table mode. In this mode, the Detail pane sorts

memory allocations by object type or size. The Graph column contains checkboxes for selecting which objects you want to graph. Disabling the All Allocations checkbox (which is enabled by default) and enabling the checkbox for another object type updates the Track pane accordingly. If you enable multiple checkboxes, the Allocations instrument layers the graphs on top of each other using different colors.

The Detail pane (while it is in Table mode) displays other useful information to help you spot potential allocation issues. The net versus overall allocations column of the table shows a histogram of the currently active objects and the total number that were ever created. As the ratio of net allocations to overall allocations shrinks, the color of the histogram bar changes. Blue histogram bars represent a reasonable ratio while colors shifted towards the red spectrum represent lower ratios that might warrant some investigation.

Although Table mode is very useful for getting an overall picture of your allocations, the Detail pane takes advantage of all three viewing modes. Table 5-3 describes the information presented in each mode and how you might use that mode to look for problems.

Table 5-3 Analyzing data in the Detail pane

Mode	Description
Table	Use this mode to see the summary of net versus overall allocations and to choose which objects you want to graph in the Track pane. Allocations in this mode are grouped by size or object type initially. Clicking the follow link button next to an object type takes you a level deeper by showing you the individual allocation events for that object. Clicking the follow link button again shows you the history of events that occurred at the same memory address.
Outline	Use this mode to see the call trees associated with allocated objects. Clicking the follow link button next to an object type focuses on the call trees associated solely with that object type.
Diagram	Use this mode to see all objects in the order in which they were allocated. Clicking the follow link button next to the object address shows the allocation events associated with that memory address.

The Extended Detail pane for the Allocations instrument primarily displays stack trace information for the selected allocation event. For some allocation events, this pane also displays descriptive information about the event, including the type and size of the event and the object retain count (if any). This information is there to help you locate the allocation event in your code.

Tracking Reference Counting Events

When you add the Allocations instrument to your document, the instrument is initially configured to record only memory allocation and deallocation events. By default, it does not record reference counting events, such as `CFRetain` and `CFRelease` calls. The reason is that recording reference counting events adds significant overhead to the data gathering process and is not needed in all situations. If your code is leaking memory, however, you might want to configure the Allocations instrument to record these events as part of your effort to track down the leaks. In particular, you can look for any mismatched retain and release events to see if an object is still retained when the last reference to it is removed.

To enable the gathering of reference counting events, open the inspector for the Allocations instrument and enable the “Record reference counts” option.

Note: The Allocations instrument found in the Leaks template document comes preconfigured with the "Record reference counts" option already enabled to help you track down memory leaks.

Filtering the Contents of the Detail Pane

Table 5-4 lists the high-level configuration options you can apply to memory events recorded by the Allocations instrument. You use these options to focus your analysis on the events that occurred at a particular time or that involve a particular portion of your code. All of these options apply only when viewing data in Outline and Diagram modes. In Table mode, the Allocations instrument displays the history of all allocations.

Table 5-4 Configuration options for the Allocations instrument

Configuration section	Description
Allocation Lifespan	Choose between displaying all allocation events and those associated with objects that still exist.
Call Tree	Choose these options to flatten or hide uninteresting parts of the call tree. You can separate out memory blocks based on which thread allocated them, hide allocations made by system libraries, show allocations made from Objective-C calls only , and more. These options help you trim irrelevant portions of the call tree and organize the remaining data in ways that make it easier to spot trends.
Call Tree Constraints	Choose the constraints for the data you want to view. You can use these configuration fields to prune the current data set. The Allocations instrument supports constraining data based on the number of allocations made for a given type or the size (in bytes) of the allocations.

When applying an instrument's configuration options, do not forget that you can also constrain the sample data based on when those samples were gathered. The Inspection Range control in each trace document lets you view the data from a specific set of sample points. This feature works in combination with all of the instrument's other configuration options. For more information on how to use the Inspection Range control, see "Viewing Data for a Range of Time" (page 50).

Looking for Memory Leaks

The Leaks instrument provides leak-detection capabilities identical to those in the `leaks` command-line tool. This instrument analyzes the in-use memory blocks of your application looking for blocks that are no longer referenced by your code. Unreferenced blocks are deemed "leaks" because they cannot be freed by your application and continue to occupy memory space until the user quits the application.

Eliminating memory leaks from your application is an important step toward improving your application's reliability. This is especially true for applications that are designed to run for long periods of time. Leaks increase your application's overall memory footprint, which can lead to paging. Applications that continue to leak memory may even find themselves unable to complete an operation because they cannot allocate the necessary memory. In extreme cases, the application may become so impaired that it crashes.

The Leaks instrument records all allocation events that occur in your application and then periodically searches the application's writable memory, registers, and stack for references to any active memory blocks. If it does not find a reference to a block in one of these places, it reports the buffer as a leak and displays the relevant information in the Detail pane.

In the Detail pane, you can view leaked memory blocks using Table and Outline modes. In Table mode, Instruments displays the complete list of leaked blocks, sorted by size. Clicking the follow link button next to a memory address shows the allocation history for memory blocks at that address, ultimately culminating in an allocation event without a matching free event. Selecting one of these allocation events displays the stack trace for that event in the Extended Detail pane along with general information about the memory block. In Outline mode, the Leaks instrument displays leaks organized by call tree. You can use this mode to find out how many leaks are in a specific branch of your code and how much memory was leaked by that branch. Selecting a branch displays the deepest code path for that branch in the Extended Detail pane.

Table 5-5 lists the configuration options for the Leaks instrument. Many of these options affect how leaks looks for information, but some affect how leaked buffers are reported.

Table 5-5 Configuration options for the Leaks instrument

Configuration option	Description
Leaks Configuration	Use the available option to enable automatic leak detection and to gather the contents of leaked memory blocks when a leak occurs.
Sampling Options	Use the specified field to set the frequency of automatic leak-detection checks.
Leaks Status	Displays the time until the next automatic leak-detection pass.
Check Manually	Use the provided button to initiate a check for memory leaks.
Call Tree	Choose these options to flatten or hide uninteresting parts of the call tree. You can hide allocations made by system libraries, show allocations made from Objective-C calls only, and more. These options help you manage the size of the call tree and organize it in ways that make it easier to spot trends.

For information about the `leaks` command-line tool, see `leaks` man page.

Analyzing Core Data Applications

For applications that use Core Data to manage their underlying data model, Instruments provides several Core Data–related instruments to analyze potential performance issues. These instruments give you insight into the events happening behind the scenes in Core Data and may help you identify places where your your application is not fetching or saving data efficiently. Table 5-6 lists the provided instruments and how you might use each one.

Table 5-6 Core Data instrument usage

Instrument	Description
Core Data Saves	Use this instrument to find a balance between saving data too often and not saving it enough. Saving too often can lead to I/O overhead as your program writes data frequently to the disk. Conversely, saving infrequently can increase the application's memory overhead and lead to paging.
Core Data Fetches	Use this instrument to optimize the data your application reads from disk. Fetch operations that take a long time might be improved by adding more specific predicates to retrieve only the data needed at that moment. Alternatively, if you notice gaps of inactivity followed by a large number of fetch requests, you might want to use those gaps to prefetch data that you know will be needed later.
Core Data Faulting	Use this instrument to track the lazy initialization of an `NSManagedObject` or its to-many relationship. Object faults can be mitigated by prefetching the object itself or the objects to which it is related.
Core Data Cache Misses	Use this instrument to locate potential performance issues caused by cache misses. Data not found in the caches must be fetched from the disk. Prefetching objects during relatively quiet periods can help mitigate cache misses by ensuring the required objects are already in memory.

The Core Data template creates a new trace document containing the Core Data Fetches, Core Data Cache Misses, and Core Data Saves instruments. This template is the recommended starting point for analyzing your Core Data applications.

For more information about tuning a Core Data application, see *Core Data Programming Guide*.

Saving and Importing Trace Data

Instruments provides a number of 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 application'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 applications can read.

Saving a Trace Document

To save a set of instruments together with the data that they have collected over one or more trace sessions, choose File > Save. Instruments saves the current document as an Instruments trace file, with the `.trace` extension.

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 application 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. Choose File > Save As Template to save your document's current instruments and configuration (including any user interface tracks) as a template.

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 that 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 applications 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. To open it, choose it from the Run > Start with Performance Tool menu.

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 applications. For example, you might save your trace data in this format so that you can import it into a spreadsheet application.

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 program'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.

To import data, choose File > Import Data. Instruments prompts you to select the text file containing the sample data. It then creates a new trace document based on the file you select.

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 75).

Creating Custom Instruments with DTrace

The built-in instruments of the Instruments application can provide a great deal of information about the inner workings of your program. 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.

Custom instruments use DTrace for their implementation. DTrace is a dynamic tracing facility originally created by Sun and ported to Mac 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 `dtrace` man page.

Note: Several Apple applications—namely iTunes, DVD Player, and Front Row, and applications 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 applications when performing systemwide data collection.

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

About Custom Instruments

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 program 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, which contains the name, category, and description of the instrument
- One or more probes, each containing its associated actions and predicates
- A DATA declaration area, which you use to declare 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 clean up actions

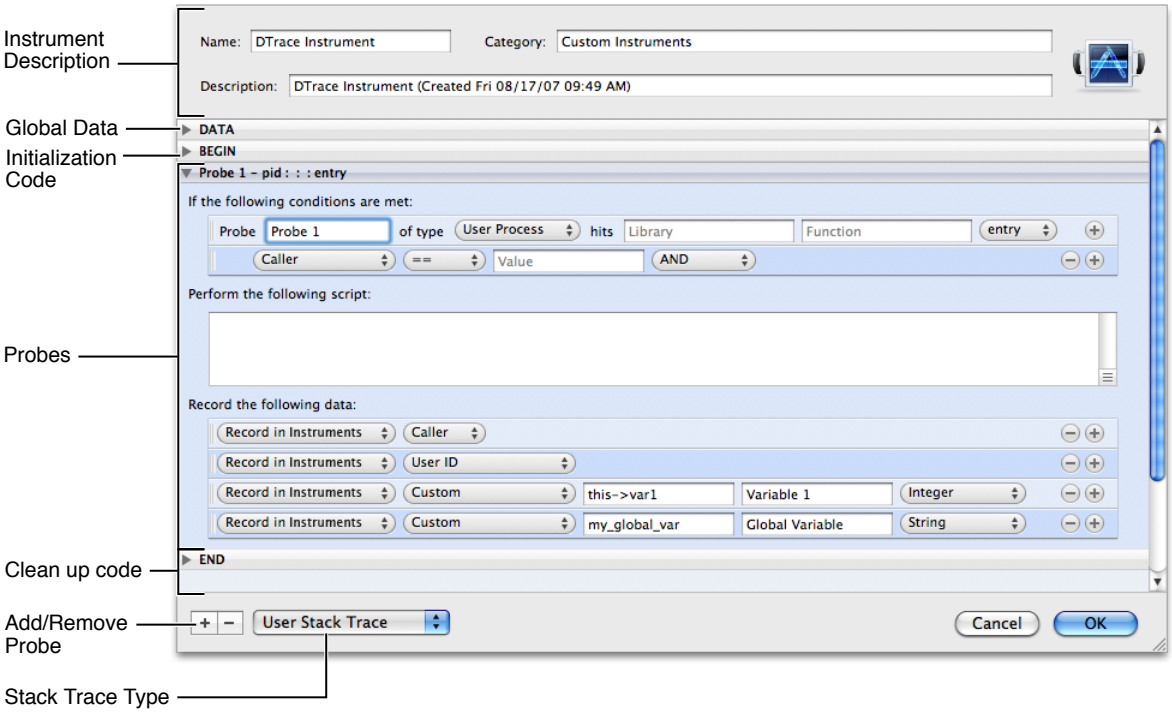
All instruments must have at least one probe with its associated actions. Similarly, all instruments should have an appropriate name and description to identify them to Instruments users. Instruments displays your instrument's descriptive information in the library window. Providing good information 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 73).

Creating a Custom Instrument

To create a custom DTrace instrument, select Instrument > Build New Instrument. (You can also choose Add Instrument > DTrace Instrument from the action menu of your trace document to perform the same action.) This command displays the instrument configuration sheet, which is shown in Figure 7-1. You use this sheet to specify your instrument information, including any probes and custom scripts.

Figure 7-1 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 and is 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 68)
- Probe action. The data to record or the script to execute when your probe fires; see "Adding Actions to a Probe" (page 71).

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 sheet.

To remove a probe from your instrument, click the probe to select it and press the minus (-) button.

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 7-1 lists the providers that are supported by the Instruments application and available for use in your custom instruments. The Provider column lists the name displayed in the instrument configuration sheet while the DTrace provider column lists the actual name of the provider used in the corresponding DTrace script.

Table 7-1 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.

Provider	DTrace provider	Description
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.
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.
Ruby	`ruby`	The probe fires at one of several Ruby-specific events.

After selecting the provider for your probe, you need to specify the information needed by the probe. For function-level probes, this may require entering function or method name along with the code module or class containing it. Some providers may simply involve selecting the appropriate events from a pop-up menu.

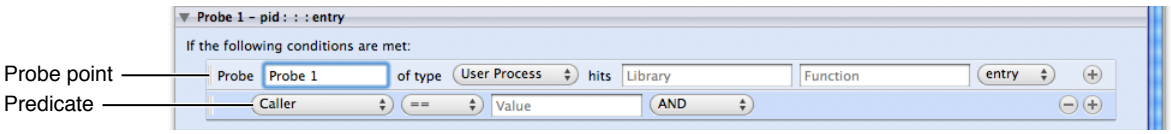
After you have configured a probe, you can proceed to add additional predicates to it (to determine when it should fire) or you can go ahead and define the action for that probe.

Adding Predicates to a Probe

Predicates let you control when a probe's action is executed by Instruments. You can use them to prevent Instruments from gathering data in situations you do not care about or that might provide erroneous information. For example, if your code exhibits unusual behavior only when the stack reaches a certain depth, you could 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, click the plus (+) button at the end of the Probe line. Instruments adds a new predicate to the probe, as shown in Figure 7-2. 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 handle along the left side of the predicate line and drag it to a new location.

Figure 7-2 Adding a predicate



Instruments evaluates predicates from top to bottom in the order they appear. You can link predicates using the AND and OR operators but you cannot group them to create nested condition blocks. Instead, you must order your predicates carefully to ensure all of the appropriate conditions are checked.

The first pop-up menu in a predicate line lets you choose the data to inspect as part of the condition. Table 7-2 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, while the DTrace variable column lists the actual name of the variable used in corresponding DTrace scripts. In addition to 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 7-2 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.
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.

Variable	DTrace variable	Description
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, measured in nanoseconds. Because this counter increments from an arbitrary point in the past, you should 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, measured in nanoseconds. This value does not include time spent in DTrace predicates and actions. This variable contains an unsigned 64-bit integer value.
Timestamp	`walltimestamp/1000`	The current number of nanoseconds that have elapsed since 00:00 Universal coordinated Time, January 1, 1970. This variable contains an unsigned 64-bit integer value.
`arg0` through `arg9`	`arg0` through `arg9`	The first 10 arguments to the probe represented as raw 64-bit integers. If fewer than ten arguments were passed to the probe, the remaining variables contain the value 0.
Custom	The name of your variable	Use this option to specify a variable or constant from one of your scripts.

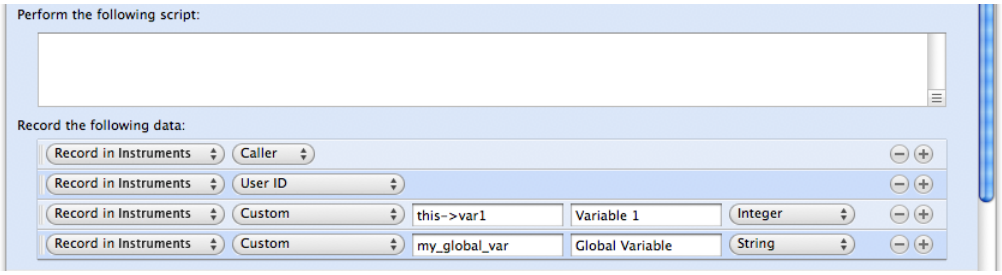
In addition to the condition variable, you must specify the comparison operator and the target value.

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 7-3 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 7-3 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 application 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 would simply wrap the variable name with the `copyinstr` subroutine as shown in Figure 7-4.

Figure 7-4 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. It does not do this with your 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 7-2 (page 70). For more information on scripts and script variables, see "Tips for Writing Custom Scripts" (page 73). 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 program.

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, however. 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 could use a script to normalize a data value to a specific range if you wanted 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 clean up 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 program 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, you must precede the name of the variable with a single back quote character (`). The back quote character tells DTrace to look for the specified variable outside of the current script.

Listing 7-1 shows a sample action script that retrieves the current load information from the `avenrun` kernel variable and uses that variable to calculate 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 7-1 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. Variables in DTrace scripts, however, can be scoped to different levels depending on your need. Table 7-3 lists the scoping levels for variables and the syntax for using variables at each level.

Table 7-3 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.

Scope	Syntax example	Description
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 the script is compiled by DTrace. 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, however, 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 application 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. Information about whether an instrument is DTrace-based is included with the instrument description in "Built-in Instruments" (page 77).

To export a DTrace script, select the trace document containing the instruments and choose File > DTrace Script Export. This 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 exported a script named MyInstrumentsScript.d, you would 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 it 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.

Built-in Instruments

A number of instruments come built into the Instruments application. Each instrument has its own configuration options and way of displaying information, appropriate to the type of data that instrument collects. The built-in instruments are grouped into a handful of categories, based on the type of information that the instrument gathers. The following sections describe the built-in instruments in greater detail.

Core Data Instruments

The following instruments gather data related to events in Core Data applications. You can use the information returned by these instruments to assess the performance implications of various events and to identify potential courses of action to correct issues.

Core Data Saves

The Core Data Saves instrument records save operations in Core Data applications. This instrument can operate on a single process or on all processes currently running on the system. It records data only for those processes that use Core Data. This instrument uses DTrace in its implementation and can be exported to a DTrace script.

Sample Data in the Detail Pane

This instrument captures the following information:

- **Caller**. The name of the method that initiated the save operation (including the stack trace information.)
- **Save duration**. The duration of the save operation in microseconds.

Display Options in the Track Pane

The Track pane can be set to display any of the following data:

- **Stack depth**. The depth of the call stack.
- **Thread ID**. The thread identifier.
- **Save duration**. The duration of the save operation.

Additional Data in the Extended Detail Pane

For each event in the Detail pane, you can open the Extended Detail pane to see the stack trace for that call, as well as the time at which the event occurred.

Core Data Fetches

The Core Data Fetches instrument records data store fetch operations in Core Data applications. This instrument can operate on a single process or on all processes currently running on the system. It records data only for those processes that use Core Data. This instrument uses DTrace in its implementation and can be exported to a DTrace script.

Sample Data in the Detail Pane

This instrument captures the following information:

- **Caller**. The name of the method that initiated the fetch operation (including the stack trace information).
- **Fetch entity**. The name of the entity that was fetched.
- **Fetch count**. The fetch count for that entity.
- **Fetch duration**. The duration of the fetch operation in microseconds.

Display Options in the Track Pane

The Track pane can be set to display any of the following data:

- **Stack depth**. The depth of the call stack.
- **Thread ID**. The thread identifier.
- **Fetch count**. The fetch count for that entity.
- **Fetch duration**. The duration of the fetch operation.

Additional Data in the Extended Detail Pane

For entries in the Detail pane, you can open the Extended Detail pane to see the stack trace for that call, as well as the time at which the event occurred.

Core Data Faults

The Core Data Faults instrument records fault events that occur during the lazy initialization of an `NSManagedObject` or its to-many relationship. This instrument can operate on a single process or on all processes currently running on the system. It records data only for those processes that use Core Data. This instrument uses DTrace in its implementation and can be exported to a DTrace script.

Sample Data in the Detail Pane

This instrument captures the following information:

- **Caller**. The name of the method that triggered the fault (including the stack trace information).
- **Fault object**. The name of the object that caused the fault.
- **Fault duration**. The execution duration for the fault handler (specified in microseconds).
- **Relationship fault source**.
- **Relationship**. The relationship name.
- **Relationship fault duration**. The relationship fault duration (specified in microseconds).

Display Options in the Track Pane

The Track pane can be set to display any of the following data:

- **Stack depth**. The depth of the call stack.
- **Thread ID**. The thread identifier.
- Fault duration
- Relationship fault duration

Additional Data in the Extended Detail Pane

For entries in the Detail pane, you can open the Extended Detail pane to see the stack trace for that call, as well as the time at which the event occurred.

Core Data Cache Misses

The Core Data Cache Misses instrument records fault events that result in cache misses. This instrument can operate on a single process or on all processes currently running on the system. It records data only for those processes that use Core Data. This instrument uses DTrace in its implementation and can be exported to a DTrace script.

Note: This instrument provides a subset of the behavior offered by the Core Data Faults instrument but is more useful for analyzing your application's overall performance.

Sample Data in the Detail Pane

This instrument captures the following information:

- **Caller**. The name of the method that triggered the cache miss (including the stack trace information).

- **Cache Miss**. The name of the object that caused the cache miss.
- **CM duration**. The execution duration for the fault handler (specified in microseconds).
- **RCM source**. The relationship cache miss source.
- **RCM relationship**. The relationship name.
- **RCM duration**. The relationship cache miss duration (specified in microseconds).

Display Options in the Track Pane

The Track pane can be set to display any of the following data:

- **Stack depth**. The depth of the call stack.
- **Thread ID**. The thread identifier.
- **CM duration**. See above.
- **RCM duration**. See above.

Additional Data in the Extended Detail Pane

For entries in the Detail pane, you can open the Extended Detail pane to see the stack trace for that call, as well as the time at which the event occurred.

Dispatch Instruments

The instrument in this section gathers data related to Grand Central Dispatch (GCD). GCD is a technology for executing asynchronous tasks concurrently. GCD is available in Mac OS X v10.6 and later and is not available in iOS.

You need to be familiar with GCD queues and block objects in order to use the Dispatch instrument effectively. For more information, see *Concurrency Programming Guide*.

Dispatch

The Dispatch instrument captures information about GCD queues created by your application and about the block objects that have been executing on these queues. It shows you the behavior of your application in terms of how your queues and block objects are executing. It records queue lifetimes, and tracks block invocations and their duration.

Dispatch helps you fine-tune the execution of your blocks. It shows you which blocks have been executing the most number of times and how long they have taken to execute on the CPU. You can find hot spots in terms of the blocks you're enqueuing and optimize your code for those blocks. You can also find cases where you have queues in which blocks are executing synchronously. (GCD queues work more efficiently when work is performed asynchronously.)

Dispatch operates on a single process. This instrument uses DTrace in its implementation, but Dispatch trace data cannot be exported to a DTrace script.

For entries in the Detail pane, you can open the Extended Detail pane to see the stack trace for that call, as well as the time at which the event occurred.

Dispatch provides several different views of the trace data. The primary views are the queues view, the call tree view, and the blocks view. The call tree view is the default view. You can use buttons located below the Detail pane to display any of the three primary views.

Queues View

The queues view shows you all the queues the application has created, the blocks that have been executed on them, and related statistics.

The queues view displays the following information:

- **Graph**. If this option is selected, the Dispatch instrument plots statistics from this queue in the Track pane.
- **Queue Name**. The name assigned by the user when the queue is created. The global queue names are assigned by the system.
- **Conc.** Indicates that this queue is concurrent (rather than serial).
- **Live**. Indicates that this queue is active (has not been freed yet).
- **# Blocks**. Number of blocks currently enqueued but not invoked.
- **# Sync**. Number of blocks that have been dispatched synchronously.
- **Total Processed**. Total number of blocks executed.
- **Latency**. Average time in milliseconds that blocks are on a queue. In other words, the average difference between invoke time and enqueue time.
- **Total CPU Time**. Total time in milliseconds that the specified queue's blocks have been executing on the CPU.

You can choose to display data from one or more queues in the Track pane. The Track pane can be set to display any of the following data:

- **Blocks Processed**. Total number of blocks processed during a specified time period. The default period is 10 milliseconds.
- **Block Count**. Number of blocks currently on a queue during a specified time period.
- **CPU Usage**. Queue CPU activity during a specified time period.
- **Work Time**. Total CPU time that a queue's blocks consumed during the specified time period.
- **Latency**. Average block latency in milliseconds during a specified time period.

For each queue listed in the queues view, you can click its focus button to see a list of the blocks that have been enqueued and invoked using the queue. This view is useful if you care about the execution order of your blocks or you want a more detailed view of the stack traces. You can select a block and display the extended detail view to see the enqueue and invoke stack traces for the block.

Call Tree View

The call tree view aggregates all stack traces and displays them as call trees. If you separate the call trees by queue, you can see which queues have had the most activity, in terms of number of blocks invoked.

The call tree view displays the following information:

- **% Calls**. Percent of total calls in which this stack trace has appeared.
- **# Calls**. Number of times this stack trace has appeared.
- **Library**. The name of the framework or bundle in which this stack trace appears.
- **Symbol Name**. The frame identifier of the stack trace.

The Track pane can be set to display any of the following data:

- **Blocks Invoked**. Number of blocks of specified type invoked during the specified time period.
- **Total Work Time**. Total CPU time that blocks consumed during the specified time period.

For each symbol in the call tree view, you can click its focus button to prune the rest of the tree and focus on that particular node and its children. You can select a block and display the extended detail view to see the heaviest stack trace for the block.

Blocks View

The blocks view displays block and queue information. This view shows all blocks executed in a queue context, including blocks that were not explicitly enqueued. For example, if block A is enqueued and executes block B, both blocks are listed in the blocks view but only block A is listed in the queues view.

The blocks view displays the following information:

- **Graph**. If this option is selected, the instrument plots statistics from this block in the Track pane.
- **Block Name**. The name assigned to this block by the compiler.
- **Block Library**. The framework or bundle in which the block has been declared.
- **Total Work Time**. Total execution time for all invocations of this block in microseconds.
- **Average Work Time**. Average execution time for this block in microseconds.
- **Count**. Number of invocations of this block.

The extended detail view for a block shows you the maximum invoked stack trace. The block has been mostly invoked from this stack trace.

You can choose to display data from one or more blocks in the Track pane. The Track pane can be set to display any of the following data:

- **Blocks Invoked**. Number of blocks of specified type invoked during the specified time period.
- **Total Work Time**. Total CPU time that blocks consumed during the specified time period.

For each block in the blocks view, you can click its focus button to see a list of queues used to execute the block. For each queue, you can click its focus button to see a list of blocks associated with the queue. This view is useful if you care about which blocks were invoked in the queue context and their order of execution.

Energy Diagnostics Instruments

The instruments in this section provide diagnostics regarding energy usage in iOS devices. They also measure the on-off state of major device components.

iOS devices behave differently when operating on battery power versus external power. This could affect the data seen in these instruments. The Energy Usage instrument, in particular, will be affected. It's not effective to measure Energy Usage while the device is connected to external power.

Here's a typical workflow to use these instruments:

1. Connect the device to your development system.
2. Launch Xcode or Instruments.
3. On the device, choose Settings > Developer and turn on power logging.
4. Disconnect the device and perform the desired tests.
5. Reconnect the device.
6. In Instruments, open the Energy Diagnostics template.
7. Choose File > Import Energy Diagnostics from Device.

Energy diagnostics data is cleared when you:

- Turn off power logging on the device.
- Disconnect the device and reboot.
- Drain the battery.

Energy Usage

The Energy Usage instrument measures energy usage since start-up. The instrument provides a macro measurement of a substantial workflow. The numeric scale is useful for comparison of different runs. Power source events (flags) are added programmatically.

The detail view displays the following information:

- **Energy Usage Level**. Relative energy usage on a scale of 0-20.
- **Power Source Events**. Transitions to battery or external power.

Note: The Energy Usage instrument is currently supported in the iPhone 3GS and the third-generation iPod touch.

CPU Activity

The CPU Activity instrument gives an indication of what the device is doing. The instrument provides a condensed version of the Activity Monitor instrumentation.

The detail view displays the following information:

- **Time**. The time interval for the measurement.
- **Total Activity**. Percentage CPU activity.
- **Foreground App Activity**. Percentage foreground app activity.
- **Audio Processing**. Percentage audio activity.
- **Graphics**. Percentage graphics activity.
- **App Activity**. Application state transitions.

Display Brightness

The Display Brightness instrument records changes in brightness that affect energy usage. The instrument does not record brightness changes due to ambient light sensor.

You can set the default screen brightness by choosing Settings > Brightness. When the screen turns on, the recording jumps to the preset level. When the screen turns off, the recording drops to zero.

Sleep/Wake

The Sleep/Wake instrument displays a red band if the device is running and darker bands if the device is sleeping, attempting to sleep, or waking from sleep. During sleep, power measurement goes to zero. This instrument is useful in correlation with other instruments.

Bluetooth

The Bluetooth instrument displays a red band if Bluetooth is active, or a black band if Bluetooth is off.

WiFi

The WiFi instrument displays a red band if WiFi is active, or a black band if WiFi is off.

GPS

The GPS instrument displays a red band if GPS is active, or a black band if GPS is off.

File System Instruments

The instruments in this section analyze file-system information and activity, such as read and write operations, permissions, and so forth.

I/O Activity

The I/O Activity instrument records I/O events: calls to functions such as `read`, `write`, `open`, and `close` that operate on files. You can use this instrument to launch and sample a single process running on an iOS device. The I/O Activity instrument is analogous to the `fs_usage` utility in Mac OS X, although I/O Activity also provides backtraces with a full call tree view.

In the Detail pane, you can choose one or more of the following categories. Each category contains a set of probes (BSD functions):

- **File Attributes**

  ```
  getattrlist
  setattrlist
  listxattr
  ```

- **File Permissions**

  ```
  chmod
  fchmod
  chown
  fchown
  lchown
  access
  ```

- **Open and Close**

  ```
  open
  fdopen
  fopen
  freopen
  close
  ```

```
fclose
```

- **Other**

```
lseek
fsync
dup
dup2
link
unlink
```

- **Read and Write**

```
read
pread
readv
write
pwrite
writev
```

- **Shared Memory**

```
shm_open
shm_unlink
```

- **Sockets**

```
recv
recvfrom
recvmsg
send
sendmsg
sendto
```

- **Stats**

```
lstat
lstat64
stat
stat64
fstat
fstat64
```

The I/O Activity instrument captures the following information:

- **Function**. The name of the function being called.
- **Duration**. The duration of the function call in microseconds.
- **In File**. The input file descriptor.

- **In Bytes**. The requested number of bytes to read or write.
- **Out File**. The output file descriptor.
- **Out Bytes**. The actual number of bytes read or written.
- **Thread ID**. The thread identifier.
- **Stack Depth**. The number of stack frames in use during this function call.
- **Error**. The most recent error during the function call.
- **Path**. The path to the file on which the executable performed the operation.
- **Parameters**. Function call parameters. For a full description of the parameters for a specific function, see the API documentation.

The Track pane can be set to display any of the following data:

- Sample number
- Call duration
- Input file descriptor
- Input bytes
- Output file descriptor
- Output bytes
- Thread ID
- Stack depth

For any function call, you can open the Extended Detail pane to see the full backtrace for that call. The instrument also provides a full call tree view in the Detail pane.

The I/O Activity instrument is sometimes used in tandem with other iOS instruments. For example, you can use I/O Activity with the OpenGL ES Driver instrument to examine the texture-loading process.

File Locks

The File Locks instrument records advisory file-locking operations that use the `flock` function call. This instrument can operate on a single process or on all processes currently running on the system. This instrument uses DTrace in its implementation and can be exported to a DTrace script.

This instrument captures the following information:

- The name of the function
- The caller of the function (including the executable name and stack trace information)
- The path to the file being locked

- The operation type, which is an integer value corresponding to one of the following values (or a combination of values):
  - 1 - shared lock
  - 2 - exclusive lock
  - 4 - don't block when locking
  - 8 - unlock

The Track pane can be set to display any of the following data:

- Stack depth
- Thread ID
- Timestamp

For entries in the Detail pane, you can open the Extended Detail pane to see the stack trace for that call, as well as any available probe information and the time at which the event occurred.

File Attributes

The File Attributes instrument records changes to ownership and access permissions for files in the file system. This instrument can operate on a single process or on all processes currently running on the system. This instrument uses DTrace in its implementation and can be exported to a DTrace script. This instrument captures information about calls to each of the following functions:

- Changes to file permission bits of a file made by the `chmod` and `fchmod` functions
- Changes to the file owner and group of a file made by the `chown` and `fchown` functions

For each function call, this instrument captures the following information:

- The name of the function
- The caller of the function (including the executable name and stack trace information)
- The path to the file on which the executable performed the operation
- The file descriptor of the file that was modified
- The mode flags, which indicate the permissions being applied to the file (This value is captured only for calls to `chmod` of `fchmod`.)
- The user ID of the new file owner (This value is captured only for calls to `chown` of `fchown`.)
- The group ID of the new group (This value is captured only for calls to `chown` of `fchown`.)

Note: For information about how to interpret the mode flags, see the `chmod` man page.

The Track pane can be set to display any of the following data:

- Stack depth
- Thread ID
- File descriptor
- Mode
- User ID
- Group ID

For entries in the Detail pane, you can open the Extended Detail pane to see the stack trace for that call, as well as any available probe information and the time at which the event occurred.

File Activity

The File Activity instrument lets you monitor file system access. This instrument can operate on a single process or on all processes currently running on the system. This instrument uses DTrace in its implementation and can be exported to a DTrace script. This instrument captures information about calls to each of the following functions:

- Open or create a file for reading or writing (`open`)
- Delete a descriptor from the per-process reference table(`close`)
- Get information about a file (`fstat`)

For each function call, this instrument captures the following information:

- The name of the function
- The caller of the function (including the executable name and stack trace information)
- The path to the file on which the executable performed the operation
- The file descriptor of the file

The Track pane can be set to display any of the following data:

- Stack depth
- Thread ID
- File descriptor

For entries in the Detail pane, you can open the Extended Detail pane to see the stack trace for that call, as well as any available probe information and the time at which the event occurred.

Directory I/O

The Directory I/O instrument records directory operations, such as moving directories, creating symbolic links, and so forth. This instrument can operate on a single process or on all processes currently running on the system. This instrument uses DTrace in its implementation and can be exported to a DTrace script. This instrument captures information about calls to each of the following functions:

- `delete` - delete files and directories
- `link` - create a hard or symbolic link
- `mkdir` - create a directory
- `mount` - mount a file system
- `rename` - change the name of a file or directory
- `rmdir` - remove a directory
- `symlink` - create a symbolic link
- `unlink` - remove a link
- `unmount` - dismount a file system

For each function call, this instrument captures the following information:

- The name of the function
- The caller of the function (including the executable name and stack trace information)
- The path to the file or directory on which the executable performed the operation
- The new file or directory name (where appropriate)

The Track pane can be set to display any of the following data:

- Stack depth
- Thread ID

For entries in the Detail pane, you can open the Extended Detail pane to see the stack trace for that call, as well as any available probe information and the time at which the event occurred.

Garbage Collection Instruments

The instruments in this section collect information on memory reclaimed by the garbage collector. To use these instruments, the program must be built for garbage collection and must be run with garbage collection enabled. See *Garbage Collection Programming Guide* for information about writing and building a program for garbage collection.

GC Total

The GC Total instrument tracks the total number of objects and bytes allocated and reclaimed by the garbage collector. This instrument can operate on a single process or on all processes currently running on the system. This instrument uses DTrace in its implementation and can be exported to a DTrace script. It records data only for those processes that have garbage collection enabled.

This instrument captures the following information:

- The function that initiated the reclamation of objects (including the stack trace information)
- The number of objects reclaimed by the garbage collector
- The total number of bytes reclaimed by the garbage collector
- The total number of allocated bytes still in use
- The total number of bytes, both reclaimed and in use

The Track pane can be set to display any of the following data:

- Stack depth
- Thread ID
- Objects reclaimed
- Bytes reclaimed
- Bytes in use
- Total bytes

For entries in the Detail pane, you can open the Extended Detail pane to see the stack trace for that call, as well as any available probe information and the time at which the event occurred.

Garbage Collection

The Garbage Collection instrument measures the reclaim data of the Garbage Collector's scavenge phase. This instrument can operate on a single process or on all processes currently running on the system. This instrument uses DTrace in its implementation and can be exported to a DTrace script. It records data only for those processes that have garbage collection enabled.

This instrument captures the following information:

- The function that initiated the reclamation of objects (including the stack trace information)
- The zone in which the memory is allocated
- Whether the event is generational (1 for yes, 0 for no)
- The number of objects reclaimed by the garbage collector
- The number of bytes reclaimed by the garbage collector
- The duration of the scavenge event (specified in microseconds)

The Track pane can be set to display any of the following data:

- Stack depth
- Thread ID
- Zone
- Is generational (the generational collector finds older objects no longer in use that are missed by the non-generational collector, but takes longer to run and is therefore run less frequently)
- Objects reclaimed
- Bytes reclaimed
- Duration

For entries in the Detail pane, you can open the Extended Detail pane to see the stack trace for that call, as well as any available probe information and the time at which the event occurred.

Graphics Instruments

The following instruments gather graphics-related data.

Core Animation

The Core Animation instrument measures the number of Core Animation frames per second in a process running on an iOS device. Offscreen frames are counted as well.

In addition, the Core Animation instrument can provide visual hints that help you understand how content is rendered on the screen. The instrument has a number of options that allow you to select specific types of rendering hints. The hints work in any application running on the device. It is not necessary to record samples to activate the hints. The hints are turned off when you close the Instruments document or delete the instrument. If there is an OpenGL surface being displayed, the rendering hints have no effect on the surface.

The instrument includes the following rendering hints:

- **Color Blended Layers**. Puts a red overlay over layers that were drawn with blending enabled. Puts a green overlay over layers drawn without blending.
- **Color Copied Images**. Puts a cyan overlay over images that were copied by Core Animation.
- **Color Immediately**. Don't wait 10 ms after performing a color-flush operation.
- **Color Misaligned Images**. Puts a magenta overlay over images whose source pixels aren't aligned to destination pixels.
- **Color Offscreen-Rendered Yellow**. Puts a yellow overlay over offscreen-rendered content.
- **Color OpenGL Fast Path Blue**. Puts a blue overlay over content that's detached from the compositor.
- **Flash Updated Regions**. Flashes updated screen regions yellow.

You can use these rendering hints to find out if you're doing unnecessary drawing—that is, redrawing content that hasn't changed.

The Track pane displays the frames per second. The Extended Detail pane shows the statistics at each sample point.

OpenGL Driver

The OpenGL Driver instrument samples OpenGL statistics. This instrument can operate on a single process or on all processes currently running on the system.

This instrument captures the following information:

- Buffer swap count
- Client GLWait time
- Command 2D Bytes per sample
- Context 2D Count
- Context GLCount
- Free Context Buffer 2D Wait time
- Gart size bytes
- Surface count
- Texture count
- Vram free bytes
- and much more...

To see the full list of captured data, install the instrument and open the Extended Detail pane.

The Track pane indicates when the data was gathered. The Extended Detail pane shows the statistics at each sample point.

OpenGL ES Driver

The OpenGL ES Driver instrument queries the GPU driver on an iOS device to sample OpenGL statistics for a single process. The instrument helps you determine how efficiently you're using OpenGL and the GPU on the device.

Note: Apple has shipped various GPUs, each with a different set of statistics. Instruments is unaware of what statistics will be displayed until it queries the device.

The GPU hardware effectively has two components: a tiler and a renderer. A scene is tiled and then rendered. The tiler and renderer components are often working on different scenes. The utilization of each component may reach 100%.

Tiler and renderer utilization can be useful for determining bottlenecks. A low renderer utilization might mean the process is stuck waiting for tiling, in which case decreasing scene complexity might help. Low tiler and renderer utilization can imply a CPU bottleneck elsewhere in a program.

This instrument captures the following information:

- **Context Count**. The number of global OpenGL contexts. Note that there are other processes running (for example, SpringBoard) that can be responsible for creating a context. This statistic is useful for keeping an eye on any errant contexts that didn't get destroyed.
- **Command Buffer Allocated Bytes**. The number of bytes allocated to store and submit command buffer data. This space is used to submit all OpenGL commands and vertex data specified by the user.
- **Command Buffer Submitted Bytes**. The number of command buffer bytes submitted to the driver. The number includes all OpenGL commands and vertex data specified by the user. Submitted Bytes is incremented for every submission, by the total amount of memory used in that submission. You may want to divide by the Submit Count value to get an idea of average per-submission usage (this average should be slightly less than Allocated Bytes, because the allocated size is bound to what is actually being used.)
- **Command Buffer Submit Count**. The number of command buffers that the driver has handled. A command buffer may contain multiple renders and transfers. Incremented when a command buffer is pushed to the GPU (each of which may contain 0 or more scenes).
- **Command Buffer Render Count**. The number of 3D frames rendered by the GPU.
- **Command Buffer Transfer Count**. The number of image operations handled by the GPU.
- **Command Buffer Swap Count**. The number of display swap commands that the driver has handled.
- **Renderer Utilization %**. The percentage of time the GPU spent performing fragment processing.
- **Tiler Utilization %**. The percentage of time the GPU spent performing vertex processing and tiling.
- **Device Utilization %**. The percentage of time the GPU spent doing any tiling or rendering work.

- **Tiled Scene Bytes**. The number of bytes used for tiling a scene. A high value indicates high scene complexity. If your scene is too complex to fit into the tiled scene bytes, you go into split scene mode, something that you should avoid. This statistic is incremented for every scene. To be useful, you need to divide the byte count by the number of scenes.
- **Split Scene Count**. The number of times a component had to go into split scene mode. Split scene mode occurs when scene complexity is high and can't fit into the tiled scene bytes buffer. Split Scene Count and Tiled Scene Bytes can be used to determine the rendering paths in which you need to reduce complexity. A split scene is something that you should avoid.
- **Resource Bytes**. The number of bytes used for textures.
- **Resource Count**. The number of textures in use.
- **Core Animation Frames Per Second**. The number of frames per second that Core Animation is compositing new frames for display. Those frames may include OpenGL ES frames from a `CAEAGLLayer` object.

The Track pane indicates when the data was gathered.

Because events are not being captured, there is no backtrace in the Extended Detail pane. Instead, the Extended Detail pane shows the full list of statistics at each sample point.

OpenGL ES Analyzer

OpenGL ES Analyzer is an iOS instrument that measures and analyzes OpenGL ES activity in an application. The instrument includes an expert system that finds problems and offers relevant solutions based on best practices and intricate knowledge of Apple's hardware and software platforms. The instrument also provides a large range of performance statistics.

Every time an application makes a call into the OpenGL ES framework, the instrument traces the call and records timing, duration, backtrace, and other parameters, and uploads this information to the host. The instrument analyzes this stream of OpenGL commands to compute useful performance statistics and to drive the expert system, that in turn offers correctness and performance suggestions.

For example, the instrument might tell you that it “detected vertex data array without using vertex buffer objects.” Vertex arrays are client data stored in main memory. It’s more efficient to upload this data to the GPU in the form of a vertex buffer object. Although the implementation is a little more complicated, the performance gains can be significant.

The instrument provides the following views:

- **Frame statistics**. Corresponds to the timeline on the graph. Gives you the data being rendered in the graph in table form.
- **Analysis findings.** Expert system recommendations, and a stack trace in the extended detail view. Color is used to indicate the severity of the problem. You can drill down to view specific occurrences of each category of recommendations, then drill down further to view which sequence of OpenGL ES commands generated that recommendation.
- **Function trace**. Entire list of OpenGL commands, with arguments and the backtrace in extended detail view.

- **API statistics**. Lists unique OpenGL calls with total time and average time for each call.
- **Call tree**. Provides navigation of all OpenGL ES or EAGL functions called by the user, leveraging Instrument's data mining tools.

When you double-click a symbol in the backtrace from the analysis findings or the function trace, the relevant line of source code is displayed.

The Overrides section is used to bypass stages in the graphics pipeline. This allows you to isolate problems and find bottlenecks in your code.

Note: The OpenGL ES Analyzer instrument is not supported in devices prior to the iPhone 3GS and the third-generation iPod touch.

Input/Output Instruments

The following instruments gather data related to I/O operations.

Reads/Writes

The Reads/Writes instrument records reads from and writes to files. This instrument can operate on a single process or on all processes currently running on the system. This instrument uses DTrace in its implementation and can be exported to a DTrace script. It gathers information about each call to read and write functions, including `read`, `write`, `pread`, and `pwrite`.

This instrument captures the following information:

- The name of the function
- The caller of the function (including the executable name and stack trace information)
- The path to the file on which the executable performed the operation
- The file descriptor of the file that was modified
- The number of bytes read or written

The Track pane can be set to display any of the following data:

- Stack depth
- Thread ID
- File descriptor
- Bytes

For any of these calls, you can open the Extended Detail pane to see the stack trace for that call, as well as any available probe information and the time at which the event occurred.

Master Tracks Instruments

The Master Tracks section contains the User Interface recorder, which lets you record and play back a series of user actions in an application.

User Interface

The User Interface instrument can launch an application or attach to a process and record your interaction with the user interface. You can then play back this recording as many times as you want, and run any other instruments you choose as you do so. You can use this instrument to create repeatable tests of the user interface as part of your quality assurance program and to capture errors that occur only sporadically. The use of the User Interface instrument is described in detail in "Working with a User Interface Track" (page 39).

Memory Instruments

The instruments in this section track memory use.

Shared Memory

The Shared Memory instrument records the opening and unlinking of shared memory. This instrument can operate on a single process or on all processes currently running on the system. This instrument uses DTrace in its implementation and can be exported to a DTrace script. It gathers information about each shared memory access, including `shm_open` and `shm_unlink`.

This instrument captures the following information:

- The name of the function
- The caller of the function (including the executable name and stack trace information)
- The name of the shared memory region
- The flags used to open the shared memory region (see the `shm_open` man page)
- The mode flags, indicating the access permissions for the region (see the `chmod` man page)

The Track pane can be set to display any of the following data:

- Stack depth
- Thread ID
- Flags

- `mode_t`

For any of these calls, you can open the Extended Detail pane to see the stack trace for that call, as well as any available probe information and the time at which the event occurred.

Allocations

The Allocations instrument tracks memory allocation for an application. This instrument requires that you launch a single process so that it can gather data from the start of the process.

This instrument captures the following information:

- Category - typically a Core Foundation object, an Objective-C class, or a raw block of memory.
- Net Bytes - the number of bytes of this type currently allocated but not yet released.
- # Net - the number of objects or memory blocks of this type currently allocated but not yet released.
- Overall Bytes - the total number of bytes of this type that have been allocated, including those that have been released.
- # Overall - the total number of objects or memory blocks of this type that have been allocated, including those that have been released.
- # Allocations (Net / Overall) - A histogram of the current and total counts. The bars are normally shades of blue. They are colored shades of yellow when the ratio between the total number of objects and the peak, or the ratio between the peak and the current number, is 1/3 or less. The bars are shades of red when the ratio is 1/10 or less.

Although the ratios displayed aren't necessarily bad (often, they're normal over the long run of an application), Instruments colors them to point out allocation patterns that may deserve a further look. If you see categories where the color is red or yellow, you might try to eliminate unnecessary temporary allocations of the given type in your application. Similarly, you might simply try to eliminate the high-water mark in the number of objects.

The data table in the details pane contains a Graph column, which contains a checkbox for each row in the table. When the checkbox for a given category is enabled, the instrument displays the graph for that particular category type in the Track pane. Instruments automatically assigns a color to each graphed category.

When you mouse over category names in the details pane, a more Info button appears next to the category name. Clicking this button displays detailed information about the objects in that category, including the following attributes:

- The address of the block.
- The function call or class that generated the allocation event. For example, you can see which method in a class retained an object.
- The creation time of the object.
- The library responsible for creating the object.

For any of these events, you can open the Extended Detail pane to see the stack trace for each object allocation, including the type of allocation and the time at which the event occurred.

For specific instances of an object (or memory block), you can click the more info button in the Object Address column to see the allocation events associated with that object. For each allocation event, this instrument displays the following information:

- The category of the object (its type)
- The event type.
- The timestamp for each event.
- The address of the block
- The size of the block
- The library responsible for allocating the block.
- The function that caused the allocation event.

For any allocation event, you can open the Extended Detail pane to see the stack trace, as well as any available event information and the time at which the event occurred.

To further filter information in the Detail pane, you can configure the Allocation Lifespan options. These options let you filter the allocation events based on the following criteria:

- All Objects Created - display all objects, regardless of whether they have been deallocated.
- Created & Still Living - display only objects that existed in memory when you stopped recording data..

The inspector for the Allocations instrument lets you configure the way the instrument tracks information. From the inspector, you can set the following options:

- Record reference counts. Use this option to track the reference count of each object.
- Discard unrecorded data on stop. Use this option to discard any data that has been gathered but not yet processed by the Allocations instrument.

For additional information about the Allocations instrument, see "Analyzing Data with the Allocations Instrument" (page 57).

Leaks

The Leaks instrument examines a process's heap for leaked memory. You can use this instrument together with the Allocations instrument to get memory address histories. This instrument requires that you launch a single process so that it can gather data from the start of the process.

This instrument captures the following information:

- The number of leaks

- The size of each leak
- Address of the leaked block
- Type of the leaked object

Each view mode in the Detail pane shows the leak data in a slightly different way. In table mode, this instrument shows the individual leaks along with the percentage that each individual leak contributes to the total amount of leaked memory discovered. In outline mode, the data is reorganized so that you can see how much memory is leaked from within a given symbol. For entries in either mode, the Extended Detail pane displays a heavy stack trace showing from where the leak originated.

For additional information about the Leaks instrument, see "Looking for Memory Leaks" (page 59).

System Instruments

The instruments in this section gather data about system activity and resources.

Time Profiler

The Time Profiler instrument stops a Mac OS X program at prescribed intervals and records the stack trace information for each of the program's threads. You can use this information to determine where execution time is being spent in your program and improve your code to reduce running time. Unlike many instruments, Time Profiler does not require DTrace probes in order to function. Time Profiler operates on a single process or all processes.

During sampling, this instrument captures the following information:

- The time at which sampling began
- The sampling duration
- Stack trace information (including the library and caller information)
- The maximum stack depth encountered during sampling
- The function encountered most frequently during sampling (the hot frame)

Time Profiler lets you view this information in different ways. In table mode, you can view the samples in the order they were gathered, which shows the execution order of your code. In outline mode, Time Profiler provides a tree view of your program's call stack and shows the number of samples that occurred in each function in that call stack.

To display a detailed call stack for a function, you can disclose items in the outline mode or select a function and open the Extended Detail pane. In outline mode, you can expand the entire call stack below a given entry by pressing the Option key and clicking the disclosure triangle for the entry.

The Track pane displays the stack depth at each sample time by default. This view is useful for identifying patterns of what is going on in your code. Because it is unlikely that two different execution paths will result in the same pattern of stack depths, when you see repeated structures in the graph, it is likely that the same code is being executed repetitively.

A unique feature of Time Profiler is the ability to record a profile without actually running Instruments. This feature is useful if you need to record a transient event and it would take too long to open and configure Instruments. To record a profile in this manner, first make sure Instruments is not running. Press the Instruments icon in the Dock. The Dock displays a menu that contains Time Profiler commands, settings, and profiles. You can choose to profile a specific process, all processes, or automatically profile any blocked (spinning) process. After a profile is created, it is listed under the Recent Time Profiles heading in the Instruments Dock menu. If you choose a profile, Instruments opens and displays the profile data.

The Time Profiler instrument and the Sampler instrument are similar, but there are some differences:

- Time Profiler gathers backtrace data in the same manner as Shark, from kernel space. Sampler, on the other hand, gathers data from user space. Consequently, Time Profiler is more efficient than Sampler at gathering data.

  Note: Time Profiler (and Shark) can yield inaccurate backtrace data if the target process is optimized to omit frame pointers.

- Time Profiler can gather data from one or all processes. Sampler can only sample a single process.
- Time Profiler can sample all thread states or running threads only. Sampler always samples all thread states. Generally, you're interested in running threads. When your application is hung, you want to examine all thread states.

Spin Monitor

The Spin Monitor instrument automatically samples any applications that become unresponsive on the system. An application becomes unresponsive when it does not retrieve events from the window server for 3 or more seconds. Applications that are unresponsive during this time may actually be doing useful work or they may be hung. You can use the sample information generated by this instrument to adjust your code so as to ensure your application keeps processing events in a timely manner. This instrument can operate on a single process or on all processes currently running on the system.

During sampling, this instrument captures the following information:

- The time at which sampling began
- The sampling duration
- Stack trace information (including the library and caller information)
- The maximum stack depth encountered during sampling
- The function encountered most frequently during sampling (the hot frame)

Each view mode in the Detail pane shows the sample data in a slightly different way. Both table and outline mode start by showing you the sessions during which a given application was sampled. Each session corresponds to a period of time where the application was deemed unresponsive and you can expand a given session to see what the application was during this time. In table mode, the instrument shows data about the functions that occurred most often during sampling. In outline mode, the instrument shows the number of samples that were gathered during each session. You can also show the running time for the samples that were gathered using the Sample Perspective options.

The inspector for this instrument lets you set the sample rate at which to gather samples. By default, this instrument gathers a sample once every 10 milliseconds.

Sampler

The Sampler instrument stops a program at prescribed intervals and records the stack trace information for each of the program's threads. You can use this information to determine where execution time is being spent in your program and improve your code to reduce running time. Unlike many instruments, Sampler does not require DTrace probes in order to function. This instrument operates on a single process.

The Sampler instrument records the following types of data for each sample:

- The function being executed
- The stack trace for each of the program's threads
- The time the sample was taken

The Sampler instrument lets you view this information in different ways. In table mode, you can view the samples in the order they were gathered, which shows the execution order of your code. In outline mode, Sampler provides a tree view of your program's call stack and shows the number of samples that occurred in each function in that call stack.

In studying the performance of a running program you should compare the impact of a function to the cost of executing that function. If your program spends a lot of time in a low-impact function, this instrument can show you that behavior. You can then use the sample data to find out why your program is spending its time there and who is calling the function, which can lead you to fixing your code so that the function is called less frequently.

To display a detailed call stack for a function, you can disclose items in the outline mode or select a function and open the Extended Detail pane. In outline mode, you can expand the entire call stack below a given entry by pressing the Option key and clicking the disclosure triangle for the entry.

The Track pane displays the stack depth at each sample time by default. This view is useful for identifying patterns of what is going on in your code. Because it is unlikely that two different execution paths will result in the same pattern of stack depths, when you see repeated structures in the graph, it is likely that the same code is being executed repetitively. If this code also takes a long time to execute, it is a good target for optimization.

For additional information about the Sampler instrument, see "Analyzing Data with the Sampler Instrument" (page 54).

Process

The Process instrument records processes forked by another process. This instrument can operate on a single process or on all processes currently running on the system. This instrument uses DTrace in its implementation and can be exported to a DTrace script.

This instrument captures the following information:

- Execute a process (`execve`)
- Process exit (`exit`)

The Process instrument returns information about each call made to these functions, including:

- The name of the function (`execve` or `exit`)
- The caller of the function (including the executable name, path, and stack trace information)
- The process ID
- The exit status of the process

The Track pane can be set to display any of the following data:

- Stack depth
- Thread ID
- Process ID
- Exit status

For any of these calls, you can open the Extended Detail pane to see the stack trace for that call, as well as any available probe information and the time at which the event occurred.

Network Activity Monitor

The Network Activity Monitor instrument records network traffic through the computer. This instrument can operate on a single process or on all processes currently running on the system.

The Track pane is set to display the following network-related data by default, but you can configure it to display other types of data as well. By default, it displays the following information:

- The number of bytes sent each second
- The number of bytes received each second
- The number of packets sent each second
- The number of packets received each second

Memory Monitor

The Memory Monitor instrument records the amount of real and virtual memory used by processes. This instrument can operate on a single process or on all processes currently running on the system.

The Track pane is set to display the following memory-related data by default, but you can configure it to display other types of data as well. By default, it displays the following information:

- The number of virtual memory page ins
- The number of virtual memory page outs
- The total amount of virtual memory space in use
- The total amount of free physical memory
- The total amount of used physical memory

Disk Monitor

The Disk Monitor instrument records disk read and write operations. This instrument can operate on a single process or on all processes currently running on the system.

The Track pane is set to display the following disk-related data by default, but you can configure it to display other types of data as well. By default, it displays the following information:

- The number of bytes per second written to disk
- The number of bytes per second read from disk
- The number of write operations processed per second
- The number of read operations processed per second

CPU Monitor

The CPU Monitor instrument records the load on the system. This instrument can operate on a single process or on all processes currently running on the system.

The Track pane is set to display the following load values by default, but you can configure it to display other types of data as well. By default, it displays the following information:

- The amount of load generated by the system
- The amount of load generated by the user
- The total load on the system

Activity Monitor

The Activity Monitor instrument records the load on the system measured against the virtual memory size. This instrument can operate on a single process or on all processes currently running on the system.

The Track pane is set to display the following load values by default, but you can configure it to display other types of data as well. By default, it displays the following information:

- The total amount of virtual memory space in use
- The amount of load generated by the system
- The amount of load generated by the user
- The total load on the system

Threads/Locks Instruments

The following instruments gather thread-related data.

Java Thread

The Java Thread instrument records the initialization and destruction of Java threads. It displays:

- The time each measurement was taken
- Total threads

You can specify the colors to use when charting the running, waiting, and blocked threads.

UI Automation

Using the Automation Instrument

The Automation instrument allows you to automate user interface tests of your iOS application. Automating UI tests allows you to:

- free critical staff and resources for other work
- perform more comprehensive testing
- develop repeatable regression tests
- minimize procedural errors

- improve development cycle times for product updates

The Automation instrument, guided by your test scripts, exercises the user interface elements of your application, allowing you to log the results for your analysis. The Automation feature can simulate many user actions supported by devices that support multitasking and run iOS 4.0 or later. Your test script can run both on the iOS device and in the iOS Simulator without modification.

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: For your protection, the instrument does not allow you to process any application that is not code-signed with your provisioning profile. This includes any copy that has been downloaded from the iTunes App Store.

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

Test Automation Script

You write your Automation tests in JavaScript, using the user interface automation API to specify actions that should be performed in your application as it runs.

Implementing automated tests with scripts can reduce development and deployment times as well as programming skill requirements for testing personnel. In addition to these benefits, JavaScript offers the sophistication to support complex operations.

See *UI Automation Reference Collection* for API details.

Your test script must be a valid executable JavaScript file accessible to the instrument on the host computer. It runs outside your application, so the tested version of your application 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"
```

Launching the Automation Instrument

With minor variations, you launch the Automation instrument much the way you would any other built-in instrument. The following procedure steps through the process:

1. Launch the Instruments application.
2. Choose the Automation template to create a trace document. (Alternatively, you can find the Automation instrument in the UI Automation group of the instrument library and drag it to your trace document.)
3. Ensure that the Detail view is displayed. (Choose View > Detail, if necessary.)

4. In the Target menu, choose a target iOS device, then choose your application from the list of iOS applications.

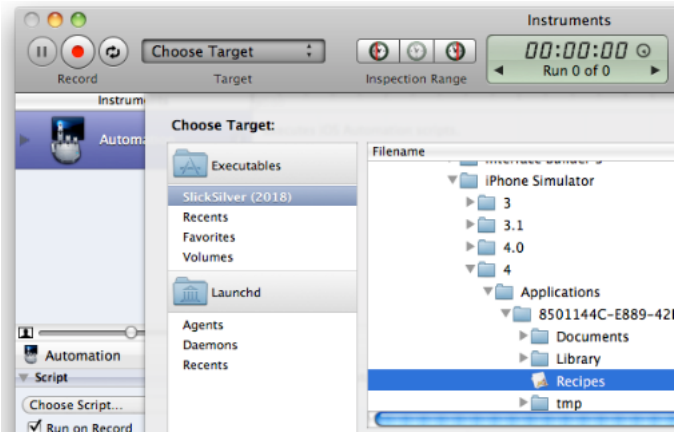
Targeting an Application in iOS Simulator

When targeting an application running in iOS Simulator, you may need to choose the target application by navigating to it in the Choose Target file browser using this logical path

```
~/Library/Application Support/iOS Simulator/<iOS version>/Applications/
```

and replacing `<iOS version>` with the actual iOS version number, for example, `4` as shown in Figure 8-1.

Figure 8-1 Targeting an application running in iOS Simulator



Upon choosing the application in iOS Simulator, two launch options become available for iOS Simulator:

- **Process I/O:** direct I/O messages to the Instruments console, the system console, or `/dev/null`.
- **Simulator Configuration:** select the combination of hardware device and iOS version to target.

Running the Automation Instrument

To run the Automation instrument, follow these steps:

1. Click the Script disclosure triangle, if necessary, to display the contents of that pane.
2. Click Choose Script.
3. In the Open panel, locate the script file and open it.
4. Click the Record button in the Instruments toolbar. Script log entries begin to appear in the Detail pane.
5. Click any script log entry in the Detail pane to reveal more information for that entry in the Extended Detail pane.

6. Use the stop and start script control buttons to stop/pause and start/resume execution of your test sequence script.

To configure Automation 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 application crashes or goes to the background, your script is blocked until the application is frontmost again, at which time the script continues to run.

Note that you must explicitly stop recording. Completion or termination of your script does not turn off recording.

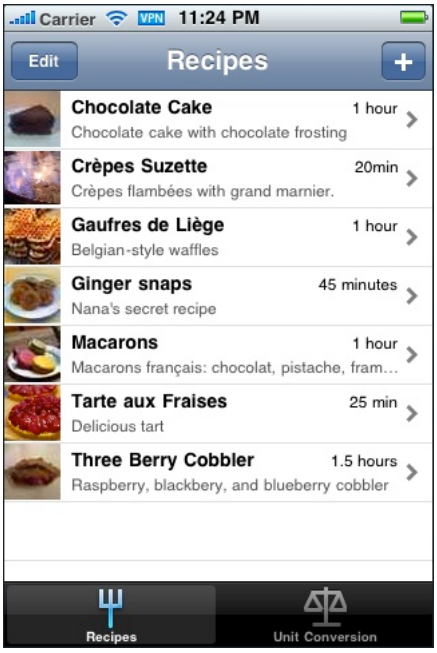
Accessing and Manipulating User Interface Elements

The Accessibility-based mechanism underlying the UI Automation feature represents every control in your application as a uniquely identifiable element. To perform an action on an element in your application, you explicitly identify that element in terms of the application's element hierarchy.

Note: 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 application shown in Figure 8-2, which is available as the code sample *iPhoneCoreDataRecipes* from the iOS Dev Center.

Figure 8-2 The Recipes application (Recipes screen)

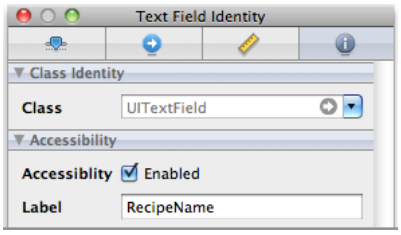


UI Element Accessibility

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 accessibility label in Interface Builder for the control represented by that element, as shown in Figure 8-3.

Figure 8-3 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 application running on that device. For the purposes of your test, your application is the frontmost application (or target application), identified as follows:

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

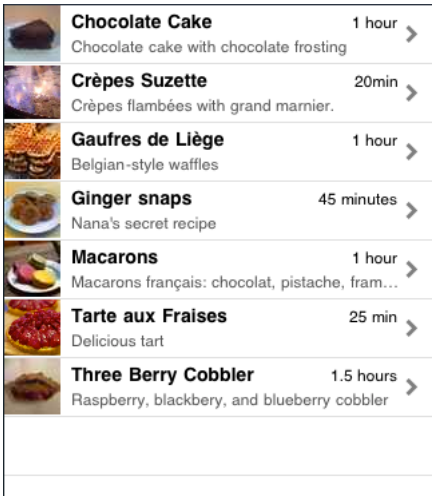
To reach the application window, the main window of your application, you would specify

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

At startup, the Recipes application window appears as shown in Figure 8-2 (page 108).

Inside the window, the recipe list is presented in an individual view, in this case, a table view:

Figure 8-4 Recipes table view

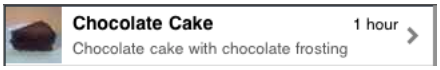


This is the first table view in the application's array of table views, so you specify it as such using the zero index ([0]), as follows:

```
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

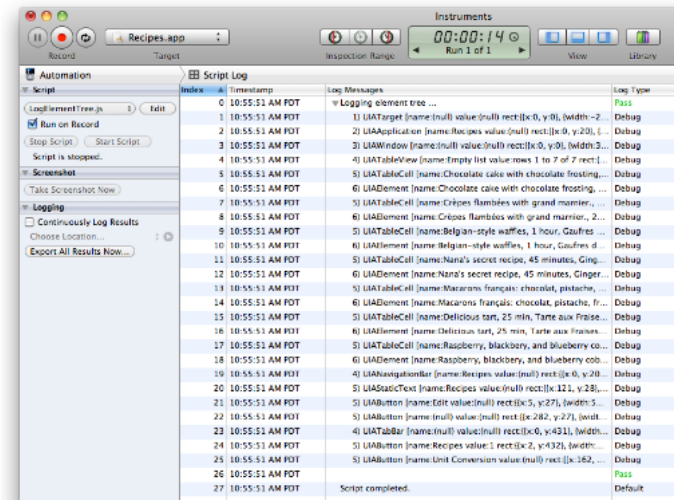
You can use the `logElementTree` method for any element to list all of its child elements. The following code illustrates listing the elements for the main (Recipes) screen (or mode) of the Recipes application.

```
// List element hierarchy for the Recipes screen

UIALogger.logStart("Logging element tree …");
UIATarget.localTarget().logElementTree();
UIALogger.logPass();
```

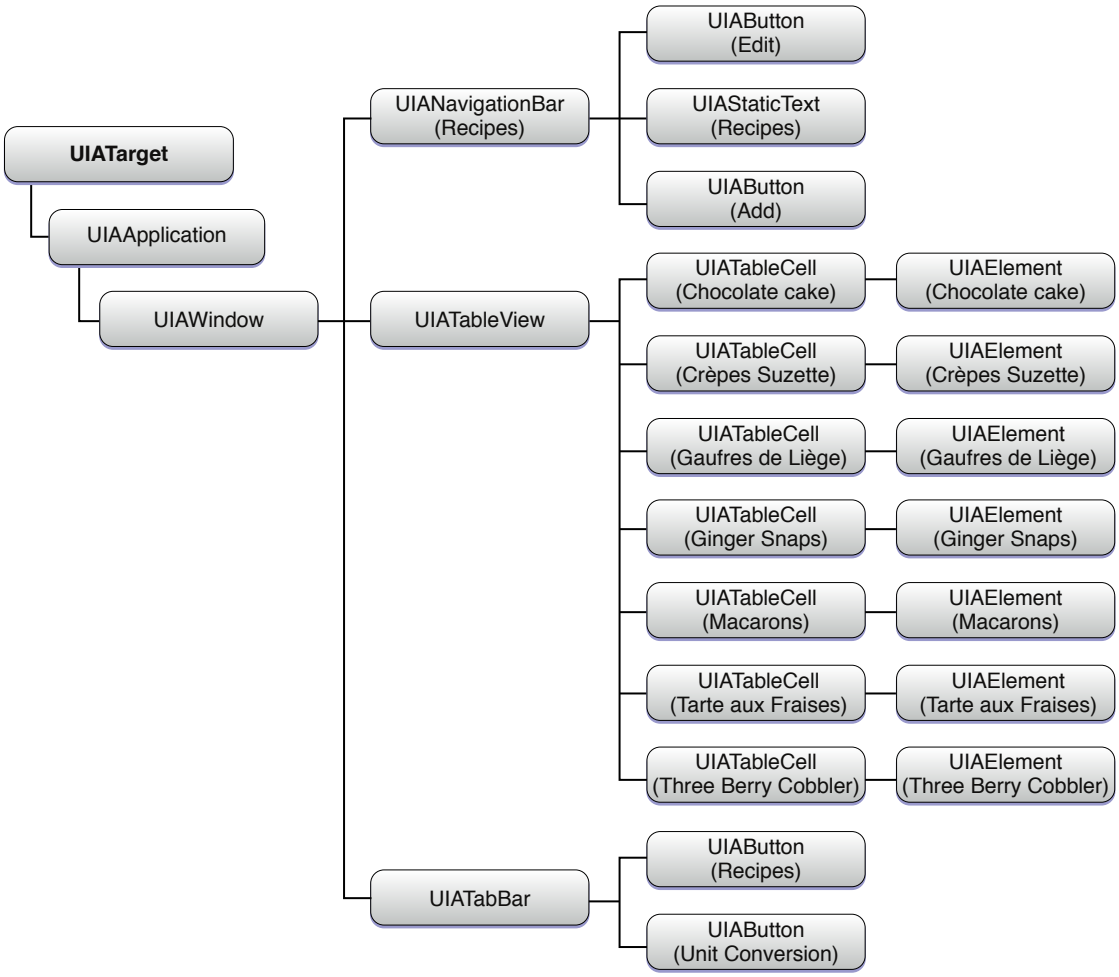
The output of the command is captured in the log displayed by the Automation instrument, as in Figure 8-5.

Figure 8-5 Output from logElementTree method



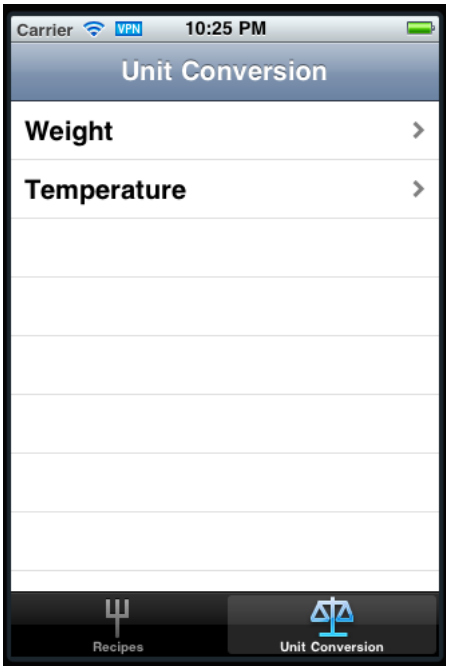
Note the number at the beginning of each element line item, indicating that element's level in the hierarchy. These levels could be viewed conceptually as in Figure 8-6 (page 112).

Figure 8-6 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 8-7.

Figure 8-7 Recipes application (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.

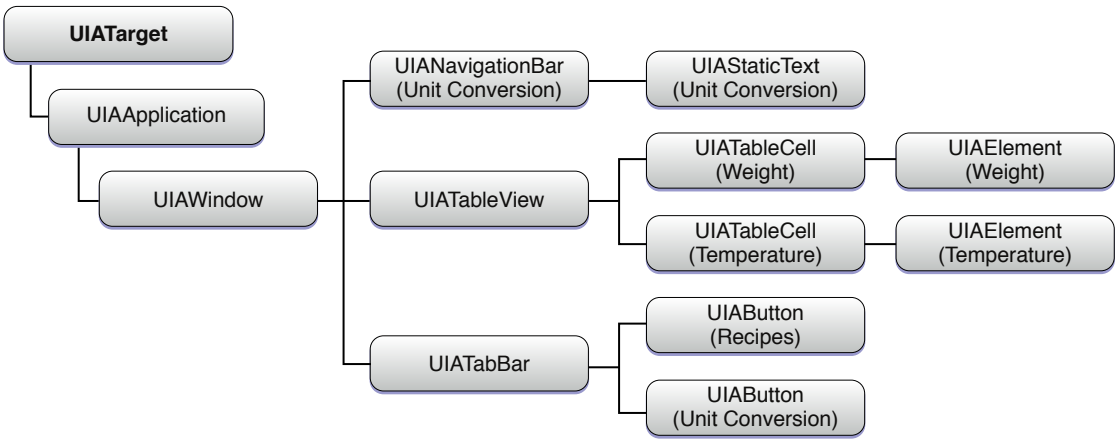
```
// List element hierarchy for the Unit Conversion screen

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 resultant log reveals the hierarchy to be as illustrated in Figure 8-8. Note that 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 8-8 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 sample uses a variable (`destinationScreen`) to control changing between the two main screens (Recipes and Unit Conversion) of the Recipes application.

```
// 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 Reference Collection*.

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 application, 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 application. 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 one 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 application 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 Application With Tabs

To test navigating between screens in your application, 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 applications. 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 application, 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 application is actually ready to accept that tap. An accurate test must ensure that the Recipes screen is completely drawn and that the application 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, it may be necessary on occasion. The following code shows how you specify a delay of 2 seconds:

UIATarget.localTarget().delay(2);

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.

Note 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 application. If, however, a user adds a recipe for Tarte aux Framboises, this example may or may not give the desired results.

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 application 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.

Handling Alerts

In addition to verifying that your application's alerts perform properly, your test should accommodate alerts that appear unexpectedly from outside your application. 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 application and your tests should expect that unexpected alerts will occur whenever your application 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 simply 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 application, you will have alerts that you do need to handle. In those instances, your alert handler will need 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, as simple as it is, 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 application 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 8-1.

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 8-1 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.

Orientation constant	Description
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 application'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 8-2.

Table 8-2 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 doing 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 application by tapping the Home button or causing some other application to come to the foreground, your application 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 application is to be suspended, as illustrated by the following example.

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

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

User Interface Instruments

The following instruments gather data for application-level events.

Cocoa Events

The Cocoa Events instrument records events sent through the `sendEvent:` method of the `NSApplication` class. This is the main method for dispatching events to a Cocoa application. You can use this instrument to correlate application events with other application behavior, such as memory and CPU usage. This instrument operates on a single process. This instrument uses DTrace in its implementation and can be exported to a DTrace script.

This instrument captures the type of event that was sent (both as an `NSEventType` code and spelled out).

The Track pane can be set to display any of the following data:

- Stack depth
- Thread ID
- The event kind

For any of these calls, you can open the Extended Detail pane to see the stack trace for that call, as well as the time at which the event occurred.

Carbon Events

The Carbon Events instrument records events returned by the `WaitNextEvent` function in the Carbon Event Manager. You can use this instrument to correlate application events with other application behavior, such as memory and CPU usage. This instrument operates on a single process. This instrument uses DTrace in its implementation and can be exported to a DTrace script.

This instrument captures the type of event that was sent.

The Track pane can be set to display any of the following data:

- Stack depth
- Thread ID
- The event kind

For any of these calls, you can open the Extended Detail pane to see the stack trace for that call, as well as the time at which the event occurred.

Document Revision History

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

Date	Notes
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.
	Added "Energy Diagnostics Instruments" (page 83) and "Using the Automation Instrument" (page 105).
2010-01-20	Added information about iPhone-specific instruments.
	Added "I/O Activity" (page 85), "Core Animation" (page 92), "OpenGL ES Driver" (page 94), and "Time Profiler" (page 100).
2009-08-20	Added information about the Dispatch instrument.
	Added "Dispatch Instruments" (page 80).
2009-07-24	Added information about gathering performance data from a wireless device.
	Added "Connecting Wirelessly to an iOS Device" (page 37).
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.

Document Revision History