

April 8, 2003

Version 2 *for Java*
on **Unix**



OC SYSTEMS

9990 Lee Highway, Suite 270
Fairfax, Virginia 22030
<http://www.ocsystems.com>

Notice

OC Systems retains all ownership rights to the programs and documentation that make up RootCause®. Use of RootCause is governed by the license agreement accompanying your original media.

Only you and your employees and consultants who have agreed to the above restrictions and those of the accompanying license may use RootCause.

You may not defeat or circumvent the license protection features built into the programs, if any.

Your right to copy RootCause and this manual is limited by copyright law. Making copies, adaptations or compilation works (except copies of RootCause for archival purposes or as a necessary part of using the programs) without prior written consent of OC Systems is prohibited.

OC Systems provides this publication “as is” without warranty of any kind, either express or implied.

Copyright © 2003 OC Systems Inc. All rights reserved.

Aprobe and RootCause are registered trademarks of OC Systems.

The trade names Ultra, Solaris, and WorkShop are property of Sun Microsystems, Inc. IBM and AIX are registered trademarks of International Business Machines Corporation. Other trademarks are property of their respective owners.

The RootCause development team includes Vince Castellano, Oliver Cole, Ivan Cvar, Dick Efron, Tom Fleck, Vasya Gorshkov, Kevin Heatwole, Paul Kohlbrenner, Steve North, Andy Platt, and Larry Preston.

Contents

CHAPTER 1	<i>Introducing RootCause</i>	
	<i>What Is RootCause?</i>	<i>1-1</i>
	<i>Java, C++, or Both?</i>	<i>1-2</i>
	<i>About This Guide</i>	<i>1-2</i>
 CHAPTER 2	 <i>Installing RootCause</i>	
	<i>Getting Help</i>	<i>2-1</i>
	<i>On-line Documentation</i>	<i>2-2</i>
	<i>System Requirements</i>	<i>2-2</i>
	<i>Reading the CD</i>	<i>2-5</i>
	<i>Installing From A Compressed Tar File</i>	<i>2-5</i>
	<i>Preparing to Install</i>	<i>2-6</i>
	<i>RootCause Console Installation</i>	<i>2-7</i>
	<i>RootCause Agent Installation</i>	<i>2-7</i>
	<i>Uninstalling RootCause</i>	<i>2-8</i>
	<i>Licensing</i>	<i>2-8</i>
 CHAPTER 3	 <i>Terminology and Concepts</i>	
	<i>The RootCause Product</i>	<i>3-2</i>
	<i>The RootCause Registry</i>	<i>3-3</i>
	<i>The RootCause Log</i>	<i>3-3</i>
	<i>Aprobe Product</i>	<i>3-3</i>
	<i>RootCause Data Management</i>	<i>3-4</i>
	<i>RootCause Overhead Management</i>	<i>3-7</i>
	<i>Glossary</i>	<i>3-9</i>

CHAPTER 4

Getting Started

<i>The Setup Script</i>	4-1
<i>The RootCause Process</i>	4-2
<i>Enabling RootCause for an AIX Application</i>	4-4

CHAPTER 5

RootCause Demo

<i>Set Up</i>	5-2
<i>Run With RootCause</i>	5-3
<i>View the RootCause Log</i>	5-4
<i>Create a RootCause Workspace</i>	5-6
<i>Define the Trace</i>	5-8
<i>Trace With RootCause</i>	5-10
<i>View The Data Index</i>	5-11
<i>Examine and Revise the Trace</i>	5-13
<i>Tracing The Details</i>	5-16
<i>Where To From Here?</i>	5-20

CHAPTER 6

Deploying the RootCause Workspace

<i>Installing The RootCause Agent</i>	6-1
<i>Building a “Traceable” Application</i>	6-2
<i>Deploying A RootCause Workspace</i>	6-2
<i>Registering a Deployed Workspace</i>	6-3
<i>Collecting Data At The Remote Site</i>	6-3
<i>Formatting and Viewing the Remotely-Collected Data</i>	6-4

CHAPTER 7

RootCause Files and Environment Variables

<i>Workspace</i>	7-1
<i>UAL File</i>	7-1
<i>XMJ File</i>	7-2
<i>Data (APD) File</i>	7-2
<i>Process Data Set</i>	7-2
<i>Deploy File</i>	7-2
<i>Collect File</i>	7-2
<i>Decollection</i>	7-2
<i>RootCause Registry</i>	7-2
<i>RootCause Log</i>	7-3
<i>.rootcause Directory</i>	7-4
<i>rootcause.properties</i>	7-4
<i>setup Script</i>	7-4
<i>Environment Variables</i>	7-5

CHAPTER 8

RootCause GUI Reference

<i>Workspace Browser</i>	8-1
<i>New Workspace Dialog</i>	8-9
<i>Reset Program Dialog</i>	8-10
<i>Add UAL Dialog</i>	8-11
<i>RootCause Options Dialog</i>	8-12
<i>Edit Source Path Dialog</i>	8-16
<i>Edit Class Path Dialog</i>	8-16
<i>Java Path Dialog</i>	8-17
<i>UAL Options Dialog</i>	8-17
<i>Java Exceptions Configuration Dialog</i>	8-18
<i>Run Program Dialog</i>	8-19
<i>Deploy Dialog</i>	8-19
<i>Decollect Data Dialog</i>	8-21
<i>Trace Setup Dialog</i>	8-21
<i>Find In Program Contents Dialog</i>	8-26
<i>Global Trace Options Dialog</i>	8-27
<i>Edit Wildcard Strings Dialog</i>	8-28
<i>Generate Custom XMJ Dialog</i>	8-29
<i>New Class Dialog</i>	8-29
<i>Trace Data Dialog</i>	8-30
<i>Add Process Data Dialog</i>	8-30
<i>Trace Index Dialog</i>	8-31
<i>Select Data Files Dialog</i>	8-34
<i>Select Events Dialog</i>	8-34
<i>Find Text In Events Dialog</i>	8-35
<i>Trace Display</i>	8-35
<i>Find Text in Trace Events Dialog</i>	8-43
<i>Table Dialog</i>	8-43
<i>Platform-Specific GUI Issues</i>	8-47

CHAPTER 9

RootCause Command Reference

<i>RootCause and Different Shells</i>	9-2
<i>rootcause</i>	9-3
<i>rootcause build</i>	9-4
<i>rootcause collect</i>	9-5
<i>rootcause config</i>	9-6
<i>rootcause decollect</i>	9-7
<i>rootcause deploy</i>	9-8
<i>rootcause format</i>	9-9
<i>rootcause log</i>	9-11
<i>rootcause merge</i>	9-12
<i>rootcause new</i>	9-13
<i>rootcause_off</i>	9-14
<i>rootcause_on</i>	9-14
<i>rootcause open</i>	9-15
<i>rootcause register</i>	9-17
<i>rootcause run</i>	9-19
<i>rootcause xrun</i>	9-19
<i>rootcause status</i>	9-19

CHAPTER 10

Selected Topics

<i>RootCause and Efficiency Concerns</i>	10-1
<i>Solaris SETUID, and Security Concerns</i>	10-3
<i>64 bit applications</i>	10-9
<i>Logging Controls</i>	10-9
<i>Multiple Application Tracing</i>	10-9
<i>Multiple Executions of a Single Application</i>	10-10
<i>Libraries with No Debug Information</i>	10-11
<i>Your Application and Different JREs</i>	10-12
<i>Using RootCause on an Application with an Embedded JVM</i>	10-12
<i>Tracing Java and C++ In One Program</i>	10-13
<i>RootCause J2EE Support</i>	10-14
<i>RootCause Shipped as Part of Your Application</i>	10-16

CHAPTER 11

Custom Java Probes

<i>A Simple Example</i>	11-1
<i>Applying One Probe to Many Methods</i>	11-4
<i>Using Method IDs</i>	11-5
<i>Logging Data from Java</i>	11-7
<i>The onLine() Method</i>	11-8
<i>Advanced Custom Java</i>	11-10

What Is RootCause?

RootCause is a sophisticated tool designed to help software organizations solve a problem as quickly as possible, ideally from a single occurrence, while simultaneously reducing support costs. Fundamental to this is a tracing capability. We have designed RootCause to make powerful application tracing and root cause analysis as simple as possible.

The fundamental concept is that all of the data needed to debug an application problem is recorded in its RootCause workspace. The RootCause Console Graphical User Interface (GUI) allows you to choose the data to be collected and to navigate the collected data.

When an application problem occurs, the “user” sends the RootCause workspace to the support organization as the problem report. If the support organization has defined the trace correctly, this RootCause workspace contains sufficient information to do the root cause analysis of the problem. There's no need to recreate the problem or ask the user further questions.

The RootCause Console tools are used in the application development and support environments to define what to trace and also ultimately to view the trace data. The application being traced may be run in the development environment as well, of course; or it may be run remotely, on a separate test platform or on a customer's computer, without access to the development environment.

You can choose to deploy the RootCause trace to the application environment after a problem occurs, or you can include RootCause as part of your shipped application so that any time a problem occurs you can immediately examine the data collected by RootCause to perform a root cause analysis of the problem.

Note that RootCause is designed to work on shipped applications. No change is needed to your application or your build processes! The traces will be automatically inserted into your application when a copy of it is loaded into memory; the traces remain only while your application is running, and they vanish afterwards.

Java, C++, or Both?

RootCause is packaged as *RootCause for Java* and *RootCause for C++*. This is the user's guide for the *Java* version only. You should read the documentation and do the demos that correspond to the version of RootCause you're interested in.

The differences in features between *RootCause for Java* and *RootCause for C++* are determined solely by the license key(s) you are issued by OC Systems. If this isn't the version you want, or you want to use RootCause on native code libraries loaded by Java, or Java run as applets or beans from a compiled application, you will need licenses to enable both the Java and C++ features. For more information see ["Licensing" on page 2-8](#), and ["Tracing Java and C++ In One Program" on page 10-13](#).

About This Guide

This User's Guide describes version 2 of the RootCause product for Java users on the Unix platform. Your feedback is desired, both on problems that you encounter and on suggestions of how the product could better accomplish its goals of solving problems from a single occurrence and of reducing support costs.

Please e-mail feedback to support@ocsystems.com and indicate what version of RootCause you are using. The version number can be obtained by with the command `rootcause help`.

If you are evaluating RootCause, or you are a first time user, we suggest that you install RootCause in a local directory (no special system administrator privileges are needed) and do the demonstrations outlined in [Chapter 5, "RootCause Demo"](#). Then return to this manual to get specific questions answered. If the information is not clear, let us know.

[Chapter 2, "Installing RootCause"](#) discusses the installation of RootCause.

[Chapter 3, "Terminology and Concepts"](#) introduces some terminology and concepts that RootCause users should know to make best use of the product. This chapter also contains a [Glossary](#).

[Chapter 4, "Getting Started"](#) discusses how an individual user would set up to use RootCause after it is installed and gives a quick description of getting started with RootCause.

[Chapter 5, "RootCause Demo"](#) demonstrates how to apply RootCause to a simple program.

[Chapter 6, "Deploying the RootCause Workspace"](#) explains how to define a RootCause trace session at your local site and then send it to a remote site to do remote debugging.

[Chapter 7, "RootCause Files and Environment Variables"](#) discusses the environment variables and files that affect RootCause.

[Chapter 8, "RootCause GUI Reference"](#) describes the RootCause Graphical User Interface in detail, briefly describing each dialog, menu item, and button.

[Chapter 9, "RootCause Command Reference"](#) describes the `rootcause` command line.

[Chapter 10, "Selected Topics"](#) contains technical discussions for issues of interest to RootCause users.

[Chapter 11, "Custom Java Probes"](#) describes how one can write probes for Java, *in Java*.

Problems and platform-specific issues are discussed in the Release Notes for the current release of the product.

Check our web site at www.ocsystems.com for white papers and the newest version of the product.

The RootCause product consists of two major components: the RootCause Console and the RootCause Agent. The RootCause Console component allows you to create probes and examine the trace data generated by the probes. The RootCause Agent is the component that performs the actual runtime tracing and generates the trace data.

Every user of RootCause will install the RootCause Console and the RootCause Agent on a local computer in order to be able to create probes and view probe data for the local computer as well as remote computers.

The RootCause Agent may then be installed on all remote computers where RootCause will be deployed (i.e. where remote applications are to be traced by the RootCause product). Note that you may also install the RootCause Console component on any and all remote computers if you wish to develop probes and view their trace data locally on the remote computers.

Getting Help

If something is missing, or you need a different media format, or you have any other installation or configuration problems, please contact OC Systems by internet at support@ocsystems.com or by telephone at (703)359-8160.

On-line Documentation

After you've installed RootCause, you can use your HTML browser and the Console's Help menu to view detailed information about use of the product.

The user guides for both RootCause and Aprobe are available in HTML format at `$APROBE/html/index.html`, and on the web at `http://www.ocsystems.com/sup_ug_index.html`.

The RootCause user guide is available in PDF, in `$APROBE/RootCauseJava.pdf`.

System Requirements

RootCause interacts very closely with the hardware, the operating system and the Java Runtime Environment on your machine. This section identifies the specific requirements in these areas. Read this carefully, and contact OC Systems if you have questions.

RootCause for Unix is currently supported on the AIX, Linux, and Solaris operating systems. On each operating system, specific compilers and Java versions are supported. Details are given below:

AIX

AIX Hardware requirements

- a POWER or PowerPC architecture workstation
- Approximately 120 megabytes of disk space for a RootCause Console installation; about 7 megabytes for the RootCause Agent alone.
- At least 128 megabytes of RAM.
- A display supporting 256 or more colors.

AIX Operating System Requirements

- AIX Version 5.1 or newer is required to run the RootCause Console Java GUI and any other tools that operate on a RootCause [workspace](#).
- The underlying Aprobe command-line facility works on AIX versions 4.2 and newer.

AIX Compiler Requirements

A C compiler is required to build *non-Java* probes. This compiler is selected at installation time and may be one of the following:

- IBM C for AIX (xlc) version 3.1 or newer.

- GCC version 2.8.1 or higher

AIX Java Requirements

Java applications must be run using Java version 1.2 or newer, which generally requires AIX 5.x. RootCause include Java version 1.3.1 for AIX, configured to enable use with RootCause as described in ["Enabling RootCause for an AIX Application" on page 4-4](#).

Linux

Linux Hardware requirements

Any modern Intel Pentium-based computer.

Approximately 120 megabytes of disk space for a RootCause Console installation; about 7 megabytes for the RootCause Agent alone.

At least 128 megabytes of RAM.

A display supporting 256 or more colors.

Linux Operating System Requirements

Red Hat Linux 7.1 or later, with a version 2.4 kernel or later.

Korn shell (/usr/bin/ksh) must be installed in order to install RootCause.

Linux Compiler Requirements

A C compiler is required to build *non-Java* probes. This compiler is selected at installation time and may be one of the following:

- GCC version 2.95.x or 2.96.

Linux Java Requirements

Java applications must be run using Java version 1.3 or newer.

Solaris

Solaris Hardware requirements

- Sparc & UltraSparc, by Sun Microsystems

- Approximately 140 megabytes of disk space for a RootCause Console installation; about 7 megabytes for the RootCause Agent alone.
- At least 128 megabytes of RAM.
- A display supporting 256 or more colors.

Solaris Operating System Requirements

- Solaris 2.5.1 / SunOS 5.5.1 or higher
- We recommend Solaris 8 or newer since that supports the preferred Java interpreter used by the RootCause Console GUI.
- For Solaris version 5.5.1, patch 103627-08 is required. Patches may be downloaded from <http://sunsolve.sun.com/>.

Solaris Compiler Requirements

A C compiler is required to build *non-Java* probes. This compiler is selected at installation time and may be one of the following:

- Sun Workshop C version 4.2 or higher; or
- GCC version 2.8.1 or higher
- NOTE: /usr/ucb/bin/cc may not be used.

Solaris Java Requirements

Java applications must be run using Java version 1.2 or newer.

Reading the CD

The CD-ROM is mounted as a file system, and once mounted is read just like a hard disk. Depending on the configuration of your system, you may need root (superuser) privileges to access or change your CD device. If you don't have access to a CD-ROM device, you can request a downloadable version from support@ocsystems.com.

AIX

On AIX, insert the Aprobe CD-ROM into the CD drive. Then, on the system containing the CD drive, mount the CD as a filesystem. If the mount is already defined (it probably is) then you can just remount it:

```
$ /etc/mount /cd0
```

Otherwise you'll have to create a directory and mount it there, which will require root privileges:

```
$ mkdir /cd0
$ /etc/mount -r -v cdrfs /dev/cd0 /cd0
```

Linux

Linux should automatically mount the CD-ROM when you place it in the drive. You should see it at `/dev/cd`.

Solaris

Solaris should automatically mount the CD-ROM when you place it in the drive. You should see it at `/cdrom/cdrom`. If you have more than one CD drive in your system you will have `/cdrom/cdrom0`, `/cdrom/cdrom1`, etc. so just pick the correct one.

Installing From A Compressed Tar File

If your CD-ROM drive is on a separate machine from where you want to install you may copy the file `rootcause_install_image.tar.Z` from the CD to disk, then ftp that to the desired machine.

Or, you may have downloaded RootCause from OC Systems directly, in a file such as `RCSol205.tar.Z`.

In either case, you may perform the following steps:

1. Copy the `.tar.Z` file where you wish to install it.
2. Execute the command:

```
uncompress -c rootcause_install_image.tar.Z | tar -xvf -
```

3. This will create a new directory, e.g., `rootcause_install_image/`. You may rename this if you prefer a different name.
4. Follow the instructions below, except that you may install “in place”, by
 - starting in the top directory (e.g., `rootcause_install_image/`), and
 - specifying “.” as the installation directory.

Preparing to Install

Once you can read the CD-ROM, the next step is to decide where to install its contents on your hard disk:

1. Examine the README file on the CD-ROM for updates to the installation process and the user guide.
2. Determine where on disk you want to install RootCause.
 - Choose a new directory which will be visible to all potential users, and which has sufficient disk space. For the full installation you will need about 140 Megabytes of space.
 - You do not need to have root privileges to do the installation unless you need those privileges to write into the selected directory.
 - RootCause should not be installed in place of an existing installation unless it is compatible (the first two digits of the version number match). Otherwise, existing probes and workspaces will need to be rebuilt.
3. Determine whether you’re using RootCause for C++ or for Java. If you’re reading this manual, you’re probably using just Java, in which case you will not need a C compiler and may skip the next step.
4. Determine which C compiler will be used during the installation.
 - See “System Requirements” on page 2-2 for a list of suitable compilers. A C compiler is required for building the probes that RootCause will create.
 - The installation script will prompt you to state or verify the full path name of the C compiler to be used during installation. This need not be the same compiler you will use to build your application(s). Rather it will be used just to compile the APC source code that describes the generated probes.
 - If a suitable C compiler is in your executable PATH, the install script will offer it as the default compiler. The GCC compiler will be cho-

sen only if no supported cc compiler is found.

5. Have a RootCause license key ready. If possible, use your mouse to copy the key from another window on your screen so you can simply paste it at the prompt during installation.

You can complete the RootCause installation process without a license key, but you'll have to install the key manually later (see "License Key Installation" on page 2-9).

You are now ready to run the installation script and reply to its prompts about the installation directory, C compiler, and license key with the selections made above.

RootCause Console Installation

RootCause is shipped on CD-ROM. You install RootCause by loading the CD-ROM and running the `install_rootcause` script found on the CD-ROM, for example:

```
/cd0/install_rootcause
```

where `/cd0` represents the CD-ROM directory described under ["Reading the CD" on page 2-5](#), or else

```
rootcause_install_image/install_rootcause
```

as described in ["Installing From A Compressed Tar File" on page 2-5](#)

It will ask you to make a few choices, including the directory location where you wish to install RootCause. It also asks for the license that was supplied by OC Systems. If you do not have a license, contact support@ocsystems.com.

If you have problems with installing RootCause, you may want to read further in this section, otherwise, you are ready to run the "RootCause Demo" on page 5-1.

RootCause Agent Installation

This section is only applicable if you want to install only the RootCause Agent component without installing the RootCause Console. This means that you wish to deploy RootCause to a remote computer, and will be creating probes for the remote computer and viewing the remotely collected probe data using the RootCause Console component located on a local computer.

The RootCause Agent installation package is contained in file `deploy/rootcause_agent.tar.Z` located on the RootCause CD and also in a RootCause Console installation.

To install the RootCause Agent on a remote computer, follow these steps:

1. Transfer file `rootcause_agent.tar.Z` to the remote computer.
2. Uncompress and un-tar the file, which will create the directory `rootcause_agent`:

```
uncompress -c rootcause_agent.tar.Z | tar -xvf -
```
3. Run the `install_rootcause` script:

```
rootcause_agent/install_rootcause
```

During RootCause Agent installation, you will be prompted for a writable directory into which the product will be installed. You may choose to install the product in place (i.e., under the `rootcause_agent` directory you just created), or install it into an entirely different directory. We recommend that you install the product on a local disk.

Note that you will *not* be prompted for a license during installation. When you create a deployable workspace (a `.dply` file) using the RootCause Console, it should contain an agent license (provided you have purchased one or more from OC Systems) that allows you to run the RootCause Agent product on remote computers.

Uninstalling RootCause

To uninstall RootCause, simply delete the entire `$APROBE` directory. The RootCause installation itself does not write to any other locations.

The `~/.rootcause` (or `~/.rootcause_aix` or `~/.rootcause_linux`) directory, and the individual workspace (`.aws`) directories, are considered to be user data, not part of the RootCause installation itself. If you delete the `~/.rootcause*` directory (or any directory referenced by the `$APROBE_HOME`, `$APROBE_LOG` or `$APROBE_REGISTRY` environment variables) you must re-run the "setup" scripts as described in [Chapter 4](#), "Getting Started".

Licensing

The accompanying license agreement describes the terms under which RootCause may be legally used. OC Systems protects its products from illegitimate use by implementing license agreement checks in its software.

Licensed use of RootCause is checked by the software using the FLEXlm licensing system from Globetrotter Software.

RootCause is packaged as *RootCause for Java* and *RootCause for C++*. This is the user's guide for the *Java* version only. The differences in features between *RootCause for Java* and *RootCause for C++* are determined solely by the license key(s) you are issued by OC Systems.

Obtaining License Keys

An demonstration license is generally provided in a cover letter or e-mail message included with the software. When you purchase the product, OC Systems will request additional system-specific information, and send you license keys generated from this information.

License Key Installation

License keys are shipped in either "decimal" or "text" format. The decimal license string can be supplied by the user when prompted during the RootCause product installation process (see "[RootCause Console Installation](#)" above).

Entering an empty license string will postpone the license installation, which will have to be done manually by editing the `$APROBE/licenses/license.dat` file and entering either the decimal or human-readable form of the license key.

For more information on the license key installation and license management refer to the FLEXlm End-User Manual that can be found in PDF or HTML format in `$APROBE/licenses` or contact OC Systems as described in "[Getting Help](#)" on page 2-1.

Terminology and Concepts

RootCause® is an extension of the Aprobe® product, a powerful general-purpose patching tool that has been in use for years. As such, much of the terminology, organization and documentation of RootCause refer to those of Aprobe.

Here we describe general terminology and concepts that apply to RootCause and Aprobe, focusing on the RootCause product. A minimal amount of Aprobe documentation is supplied here, just enough to support the RootCause definitions. For additional information, see the Aprobe user's guide (\$APROBE/Aprobe.pdf).

The RootCause Product

We use the terms *application* and *program* interchangeably throughout the RootCause product. An application or program is represented by a Java main class.

We use the term *probe* when describing what RootCause does: RootCause “probes” a running application. The probes created by RootCause do things like tracing, timing, data collection and more. Note that these probes are added only to the in-memory copy of the running application; RootCause does not modify the disk-resident application at all.

Each application that is probed by RootCause is assigned a *workspace*. A workspace is a directory where RootCause can put all of its important files (including the data collection files) at runtime.

Each workspace is created and initialized only once, when RootCause is first invoked on an application. Thereafter, RootCause automatically manipulates the workspace contents, so users can ignore the workspace during normal use. For each probed application, there is one workspace; and for each workspace, there is one application.

We use the term *log* as a verb to describe Aprobe's low-overhead data recording mechanism. RootCause logs its data into files in the workspace. For best performance, workspaces should be on a local disk (not remotely mounted).

The RootCause product is invoked by the command `rootcause` (see [Chapter 9, "RootCause Command Reference"](#)).

The RootCause product consists of two major components: the *RootCause Console* and the *RootCause Agent*. The RootCause Console component allows you to create probes and examine the trace data generated by the probes. The RootCause Agent is the component that performs the actual runtime tracing and generates the trace data.

You can choose to install only the RootCause Agent on a remote computer and then use the RootCause Console's *deploy* operation to create a workspace for that remote computer. The deployed workspace can then be transferred to the remote computer for use by the RootCause Agent there, and the trace data can be transferred back to the RootCause Console for examination.

The RootCause Registry

The RootCause Agent is enabled on a per-process-group basis via an environment variable. When rootcause is “on” in your environment, RootCause will identify and optionally record the creation of every new process and subprocess created in that shell or subshells inherited from that shell in the RootCause [log](#). If the main class for the process is listed in the RootCause [registry](#), then RootCause will insert probes into that process to collect data the next time it is launched (when you registered the program, you specified the workspace associated with the main class and that workspace contains the probes). If the main class does not appear in the registry, then RootCause allows the process to continue undisturbed.

The RootCause Log

The RootCause log is the central reporting place for RootCause. By default, RootCause records every process that is started while “rootcause_on” is in effect. RootCause also writes error messages to this file. The default behavior when starting the RootCause Console [GUI](#) (with the “rootcause open” command) is to view this log, from which you may read error messages, open workspaces, and view trace results.

The log is a fixed size, and wraps around to avoid growing too large. The [root-cause log](#) command manages attributes of this file.

Aprobe Product

Since RootCause is an extension of the Aprobe product, the setup scripts define an environment variable, APROBE, which identifies the RootCause installation directory. \$APROBE is used throughout this manual to refer to the installation directory of the Aprobe and RootCause products. This section introduces some Aprobe terminology.

As your program executes, tracing data is logged (i.e. written) to an APD (Aprobe Data) file. Almost always, more than one APD file is allocated, and these files are used in a round-robin fashion (the oldest APD file is always overwritten). This set of APD files is referred to as an *APD ring*. There is a separate control file that is used to manage all the files in the APD ring; this control file is named the APD ring file.

APD files are written in a proprietary binary format. The `apformat` command reads APD, UAL and program files and generates readable text.

RootCause Data Management

Tracing an application raises a number of questions about managing the data that is recorded.

1. How can trace data be recorded quickly?
2. How can the least amount of data be recorded?
3. How can data recorded by multiple instances of the same program be preserved and organized?
4. How can the total amount of data recorded be bounded, to keep from filling up the disk?
5. How can users “snapshot” important data to be kept, while still bounding the total data collected?
6. How can users find what they’re looking for in the data that is collected?

To address these issues, RootCause provides an interface to the Aprobe “log” mechanism which provides powerful and flexible data-recording and formatting capabilities. Here’s how they work.

Recording Data Quickly

Aprobe logging uses memory-mapped files to record the data. Each process has its own set of data files mapped to distinct memory regions, which avoids bottlenecks and locking problems when several processes are logging data simultaneously. Thread-safety is managed primarily using a lock-free compare-and-swap mechanism, though some locking is still required when switching data files.

However, even though the files are memory mapped, the contents must eventually be written to disk, and this is limited by I/O speeds to the disk device. For this reason, it is very important that the workspace directory (where RootCause writes the data files) be “local” (directly connected to the machine where the traced program is running) and not accessed across the network (e.g., using NFS). If you are collecting data from a program running on multiple machines using the same workspace, or have other special requirements, contact OC Systems.

Recording Less Data	The Aprobe “log” mechanism automatically separates the recording of data from its display and formatting. For example, timestamps are recorded simply as 64-bit values at run-time, then formatted as desired later. String literals and labels are also added as part of formatting. This has the added benefit of being able to format the same data in multiple ways without rerunning the application.
Data for Multiple Processes	The data associated with each process is saved in a “ Process Data Set ”, or “ APD ring ”, a directory identified by the PID of that process. The user specifies how many of these should be saved, and when that limit is reached, the oldest Process Data Set is deleted when tracing of a new process is started.
Bounding Total Data	<p>As mentioned above, the user specifies how many processes’ data are to be saved. In addition, the amount of data saved for each process is also highly configurable. The data for each process is treated as a multi-file circular buffer, or “APD ring”. Each file is called an “APD file” because of its suffix, “.apd”. At the Aprobe level the user may specify the size of each file, the number of files in each ring; and the number of snapshot files saved. RootCause makes this a bit easier by offering the following parameters in the RootCause Options Dialog:</p> <ul style="list-style-type: none">• Keep logged data for N previous processes This specifies the number of Process Data Sets to keep, as described above.• Data File Size (bytes) This specifies the size of each “APD file”.• Wraparound data logging wraps at N (bytes) This specifies the total “wraparound buffer” size, which corresponds to the number of individual data files that are kept for each process before the oldest start being deleted.• Total logged data limit per process (bytes) Files may be preserved even when they might otherwise be deleted, using a snapshot mechanism described below. This parameter allows the user to set a hard upper bound on the total data per process, even when many snapshots are taken.

Data Snapshots

RootCause provides a mechanism for a “snapshot” to be taken programmatically. This does not copy the data, but rather marks it as “preserved” so it is not deleted by the normal wraparound mechanism described above. RootCause allows a user to identify points during program execution at which a snapshot of the data is to be taken. At the Aprobe level, users may specify arbitrarily complex conditions under which a snapshot is taken. This mechanism is used by the [java_exceptions](#) predefined UAL, which causes a snapshot to be taken when selected Java runtime exceptions occur.

Data Indexing

RootCause provides four levels of control in accessing the data:

- the [Process Data Set](#);
- individual [Data Files](#);
- special events in the [Trace Index Dialog](#); and
- individual events in the [Trace Display](#).

Data is generally selected via the Trace Index Dialog, by double-clicking on a Process Data Set in the Workspace Tree, or by clicking the Index button. Index entries are shown for the “Last Data Recorded”, and for any snapshots taken. In addition, any exceptions detected by the [exceptions](#) and [java_exceptions](#) predefined UALs may be shown in the Index. (We anticipate additional kinds of events being available through the Trace Index in future versions.) One or more events may be selected in the Index, and a Trace Display opened on the files surrounding that event. You can control the size of the context around the event via the [RootCause Options Dialog](#).

From the Trace Index Dialog you can specify what Data Files the Index is to be generated from, and you can add data files from additional processes, and even additional workspaces. Using the Examine button in the Workspace Browser you can directly specify which Data Files are to be viewed, without going through the Index.

Once you have selected which data files to view, you can view all the data collected in the [Trace Display](#). This shows all the events ordered by thread or by-time, and organized as a call tree within each thread. One can do textual searches through this display for specific events. Data may be added or removed from a Trace Display at the Data File level, and the RootCause Log may be merged with it as well to show the interaction of multiple processes.

RootCause Overhead Management

After a program has started, the overhead that a RootCause trace adds is proportional to the number of traced method calls made by the program as it's running. Often it's the case that the most-frequently-called methods are of little interest in the trace, and yet are introducing the most overhead.

RootCause provides several mechanisms to control tracing overhead, and focus the tracing to the time when it will provide the most information.

Load Shedding

RootCause manages tracing overhead by [load shedding](#), a process by which tracing is disabled based on its estimated tracing overhead. This is done automatically by default, based on a heuristic analysis of CPU time used by traced functions. You can disable load shedding, or adjust the heuristics, with the [Global Trace Options Dialog](#) opened by clicking the Options button at the bottom of the [Trace Setup Dialog](#).

When viewing the data, if there are any functions that were load shed, a `LOAD_SHED` node will appear in the [Event Trace Tree](#), from which you can open a [LOAD_SHED Table](#) to see exactly which functions were disabled and when.

Using this table you may change the disposition of some or all of the functions during the next run. Usually you will simply want to select all the functions listed and change them to "Don't Trace" so they aren't traced at all in subsequent runs. However, occasionally a function will be disabled that is important to trace, and in this case you may mark that function as "Don't Shed" to force it to be traced regardless of the overhead.

Traced methods designated as "Don't Shed" are marked with a red dot in the [Trace Setup Dialog](#). You can enable or disable load shedding on a specific method using the [Trace Setup Popup Menu](#).

Enable/Disable Tracing

RootCause provides a mechanism to disable tracing at the start of the program, (or any other point) and enable it upon entry to (or exit from) a method executed later on. This is conceptually a global switch that can be turned on and off during program execution. So, for example, if your program does a lot of processing during initialization, but you're not interested in tracing that, you can:

Select the Program node in the [Trace Setup Dialog](#), and using the [Probes Pane](#) configure a Probe On Program Entry to Disable Tracing.

Then you can select a method that is called at the start of the processing you want to trace and create a Probe On Entry to Enable Tracing.

As with most features available through the RootCause console, you can get even more control and power with a custom probe which directly calls the Aprobe runtime functions `ap_RootCauseTraceEnable()` and `ap_RootCauseTraceDisable()` to enable and disable tracing only if certain conditions or data values are detected in the program. Contact support@ocsystems.com for more information on writing custom probes.

Glossary

Discussion of RootCause and Aprobe requires the use of terms that are either specific to the products or assigned a special meaning in the context of the product. Many of these terms are also defined above and elsewhere in this guide, but are listed here for easy reference.

ADI file: Aprobe Debug Information file, which contains symbol and line information extracted from a [module](#) for use by [deployed](#) probes on [applications](#) which have been [stripped](#) of debug and symbol information.

APC: “AProbe C” language, a superset of the C programming language, used to define [probes](#). An APC file is a text file containing APC source code. APC files are compiled into a [UAL](#) file using the `apc` command.

APD file: “AProbe Data” file, containing information written in a compressed format by the `log` command. These files are [formatted](#) (generally converted to text) by the `apformat` command.

APD ring: A set of [APD files](#) corresponding to a single execution of an [application](#). There is always one persistent file, “name.apd”, and one or more ring files “name-1.apd”, “name-2.apd”, etc., grouped together in a directory having the same name as the persistent file, but with a suffix corresponding to the PID of the traced process, e.g., “name.apd.12345”.

actions: Operations, generally gathering or counting data, that are applied at a certain point in a program. Actions, combined with the points where they are applied, make up [probes](#).

agent: The part of the RootCause product which actually applies and enables the [probes](#), also known as the Aprobe runtime.

apformat: The Aprobe command which [formats](#) (generates text from) [APD files](#).

application: An executable or JRE together with all the classes or shared libraries that it loads, also known as a [program](#).

aprobe: The Aprobe command which actually applies [probes](#) to a [program](#).

collect: To identify the [APD files](#) from one or more [workspaces](#) and compress them, along with other necessary information, into a single file with the suffix

.clct, usually in preparation for moving from a remote machine back to a local [Console](#) installation for analysis.

Console: The RootCause [GUI](#) and supporting Aprobe tools (e.g., `apc`, `apformat`), through which all development and analysis of traces is performed.

Data File: A file containing RootCause data logged when a process is run under rootcause; another name for an [APD file](#).

decollect: To expand a `.clct` file back into a directory with suffix `.dclct` for analysis by the RootCause [Console](#).

decollection: the `.dclct` directory tree created by the [decollect](#) operation.

deployment descriptor file: An [XML](#) file with suffix `.xmj`, which specifies the conditions under which a Java probe is to be applied, and what information that Java probe needs. See [Chapter 11, "Custom Java Probes"](#).

dynamic module: A class or jar file from which classes are explicitly loaded after execution has begun. See ["Add Dynamic Module" on page 8-5](#).

deploy: To compress the information in a [workspace](#) into a file with a `.dply` suffix, and transfer that file to a [remote](#) computer for tracing an [application](#) there.

event: any of a number of specially-tagged data items [logged](#) by RootCause and shown in the [Trace Index Dialog](#) or [Event Trace Tree](#), or printed by the root-cause format command.

executable: A binary object file containing the entry point of an [application](#) which may be run directly; this is distinct from a [shared library](#), which must be loaded in the context of a running executable.

format: To process the contents of [APD files](#) into text, or other meaningful output, using the Aprobe `apformat` command. Data collected by RootCause is generally formatted into [XML](#) before being read into the RootCause Console.

GUI: The Graphical User Interface portion of the RootCause [Console](#). See [Chapter 8, "RootCause GUI Reference"](#).

JRE: “Java Runtime Environment”, the environment which directly executes Java applications. The RootCause [GUI](#) is implemented in Java and so ships with a JRE. RootCause for Java allows you to define Java traces for supported JREs.

JVM: “Java Virtual Machine”, the portion of an application (e.g., java, Netscape) which loads and executes Java class files and applets. This is generally implemented as a single library within the [JRE](#).

load shedding: The process of dynamically disabling tracing of functions or methods based on the tracing overhead they introduce into the program. This mechanism prevents tracing from slowing down a program too much, and automatically creates a list of methods to be eliminated from subsequent traces.

local: Referring to the machine and execution environment in which the RootCause [Console](#) is installed, where [traces](#), and perhaps the traced [application](#) itself, are developed; the opposite of [remote](#), where the [agent](#) is installed.

log: *verb:* the recording of data by RootCause into an APD file.

log: *noun:* the RootCause Log, in which information about processes is recorded.

module: A loadable object module; an [executable](#) or [shared library](#). In RootCause for Java, a class and all its supporting classes are managed as a module as well.

PID: Process ID, the number assigned to each process on the system, and used to uniquely identify each [APD ring](#) generated by [tracing](#) that process.

Process Data Set: The group of [Data Files](#) associated with a single process ([PID](#)); another name for an [APD ring](#).

predefined UAL: A ready-to-use [UAL](#) which may be applied to any application to perform a specific function. Some are provided with RootCause, additional ones with Aprobe, and more can be developed by users.

probes: Actions to be performed at specific points in a [program](#). These [actions](#) are applied at the probe points in memory, without modifying the program files on disk.

program: An executable or JRE together with all the classes or shared libraries that it loads, also known as an [application](#).

register: To associate a [program](#) with a [workspace](#), so that [tracing](#) will occur when the program is run with rootcause on.

registry: The database mapping [programs](#) to [workspaces](#), and recording other information that must be checked when programs are [run with rootcause on](#). This is implemented as a text file named by the environment variable APROBE_REGISTRY.

remote: Refers to a machine or execution environment separate from that in which an [application](#) is developed; the opposite of [local](#). In a remote environment, the [modules](#) that make up a [program](#) may be fully or partially [stripped](#), and the [workspace](#) in which the probes were developed is not accessible, so the workspace must be [deployed](#).

run with rootcause on: To execute a [program](#) in an environment where RootCause is intercepting processes. This is generally done by running the `rootcause_on` command, then running the application. (On AIX, use the [root-cause run](#) command; see "[Enabling RootCause for an AIX Application](#)" on page 4-4). Some simple applications may be run directly with the Rootcause GUI Run button.

shared library: A linked object file which may be shared by many programs, but cannot be run by itself.

shadow header file: A C header file which provides debug information for the system library of the similar name. For example, `$APROBE/shadow/libc.so.h` is a shadow header for `libc.so` on Solaris

snapshot: A copy of data saved at the point of a notable event. In the context of RootCause, `SNAPSHOT` [probes](#) may be inserted which ensure that the associated data is preserved.

stripped: An application which was built with debug and symbol information, but from which that information has subsequently been removed (such as by running the `strip(1)` command), is referred to as a “stripped” application.

timestamp: a string indicating the “wall-clock time” at which an [event](#) occurred.

traceback: A display of the call stack starting with the function/method in which the traceback was generated, followed by its caller, then its caller's caller, etc.

traces: A subset of probes which quickly record the entry and exit of identified functions or methods. These are indicated in the [GUI Trace Setup Dialog](#) by black dots next to the entities containing traces, as distinct from [probes](#).

tracing: The process of applying the [traces](#) and [probes](#) in a RootCause [workspace](#) to an [application](#). We use this term in general to refer to the data-gathering that takes place when a [registered](#) application is running with rootcause on.

trigger: The point at which an action takes place. In particular, when defining probes within the [Probes Pane](#), it may be the entry or exit of a program, thread, or method.

UAL: “User Action Library”, a [shared library](#) defining “user [actions](#)” or [probes](#), and the program points to which they are applied.

workspace: A directory with suffix `.aws` created and managed by the RootCause [GUI](#) and `rootcause` command, which contains the [traces](#) and [probes](#) on an [application](#), the [APD rings](#) generated from those, and scripts for [formatting](#) that data.

XMJ file: See [deployment descriptor file](#).

XML: “eXtended Markup Language”, a text language for expressing hierarchical information. RootCause [formats](#) the [APD files](#) collected by [tracing](#) into an informal XML syntax which is then consumed by the [GUI](#).

The Setup Script

Installation of RootCause from CD-ROM or compressed file is covered in [Chapter 2, "Installing RootCause"](#).

After installation, but each time before you run RootCause, you will need to execute its setup script. This will set the necessary environment variables (e.g. APROBE, PATH, etc.). This is typically done by each user's login script because it usually needs to be done only once per login session.

The RootCause installation automatically creates a setup script that must be executed before RootCause can be run. This script is located in the root of the directory where RootCause was installed.

For example, if you installed RootCause in directory /opt/aprobe, then you would execute the appropriate one of the following scripts to set up the environment before using RootCause. The first of these scripts is for ksh or bsh users; the other is for csh users:

```
. /opt/aprobe/setup
```

or

```
source /opt/aprobe/setup.csh
```

You must execute this setup script in your shell every time you log in or otherwise reinitialize your environment. Therefore it is a good idea to put the appropriate command above in your `~/.profile` (for Korn or Bourne shells) or `~/.login` file (for C shell). See “RootCause and Different Shells” on page 9-2 for information about other shells.

This script defines the `APROBE` environment variable and appends `$APROBE/bin` to your `PATH` environment variable. It also sets defaults for both `APROBE_REGISTRY` and `APROBE_LOG` environment variables. If the registry does not exist, the setup script will create a default one.

The setup script also defines aliases that are not inherited by subsequent non-login shells you may open, such as with the `xterm` command. To ensure that these aliases are defined (specifically `rootcause_on` and `rootcause_off`), you may add the command:

```
. $APROBE/setup.kshrc
```

into your `~/.kshrc` file if you are using Korn shell, or the command

```
source $APROBE/setup.cshrc
```

into your `~/.cshrc` file if you are using C shell.

Now you're ready to run RootCause.

The RootCause Process

Using RootCause is typically an iterative process with the following pattern:

1. Run your application in the normal way, but with RootCause enabled in its environment, for example:

```
rootcause_on
java -classpath . TestDriver arg1 arg2
rootcause_off
```

This will record information about the application in the [RootCause Log](#) file.

NOTE: On AIX, you must use “`apaudit java`” in the above command instead of just “`java`” to allow RootCause to log and trace your application. See [Enabling RootCause for an AIX Application](#) below.

2. Start the RootCause Console GUI with the `rootcause open` command. This will open the [Workspace Browser](#) and the [Trace Display](#) showing the contents of the RootCause log file.

3. Use [Open Associated Workspace](#) on the application program listed in the RootCause log display, and approve the creation of the new workspace.
4. Click [Setup](#) and define the probes for the application by choosing what you want to trace in the [Trace Setup Dialog](#). Note that RootCause writes the probes to separate files, so the application itself remains totally unmodified.
5. Execute the application as you normally would, but with rootcause “on”, as in step 1 above. The application need not be executed from the RootCause GUI, although there is a convenience [Run](#) button to do so. This makes it very simple to use RootCause, even if the application is deeply embedded in a complex system.
6. Click the [Index](#) button to bring up the index of data that was logged for the newest process in the workspace, and double-click on an item there to format the trace data collected by the probes, or you can use the [Examine](#) button to directly select the data file(s) you wish to view.
7. In the [Trace Display](#) window that appears is a call tree. Here you can:
 - Use the [Find](#) button to search for specific functions or events in the data.
 - Select a node in the event tree and right-click to bring up the [Trace Display Popup Menu](#), from which you can disable traces and perform other operations.
 - Select a `_SYN_JAVA_CALL_COUNTS` node and use [Show Associated Table](#) to view and navigate to the methods in the data file(s).
 - Select a `JAVA_LOAD_SHED` node and use [Show Associated Table](#) to view the table of all methods for which tracing was disabled due to [load shedding](#) during the run, and disable the tracing of these during subsequent runs.
 - Save the output as Text or XML for off-line processing
8. When you have completed analyzing the data and modifying the trace to tune the information collected, go back to step 4 to trace parameters or add probe actions, or return to step 2 to choose another program to trace.

If you wish to run the probes on a remote computer, then there are additional steps to send a RootCause workspace to the remote computer before execution. These are discussed in detail in [Chapter 6, "Deploying the RootCause Workspace"](#).

This process is explored using a concrete example in the next chapter, "[Root-Cause Demo](#)".

Enabling RootCause for an AIX Application

AIX provides no mechanism to cause a shared library to be automatically loaded into every application started. Therefore, the user must explicitly identify applications that are to be “intercepted” and recorded in the RootCause log, and subsequently traced using the process described in this document.

To run an executable under RootCause from the command line:

```
$ rootcause run my_program -opt arg2
```

or

```
$ rootcause on  
$ apaudit my_program -opt arg2  
$ rootcause off
```

If it is not possible to directly invoke the application’s executable from the command-line, you can simply rename the [executable](#) to have a .exe suffix, and replace it with a (soft-link to a) script which will invoke the real application. For example:

```
$ mv my_program my_program.exe  
$ ln -s $APROBE/bin/run_with_apaudit my_program.exe
```

The `run_with_apaudit` script simply re-invokes the program with `apaudit` as in the second example above.

When we use the phrase “[run with rootcause on](#)” we refer to any of the above mechanisms.

To enable automatic RootCause actions for Java applications (i.e., those normally run with the “java” command), you must do this for the native `java` executable. This is generally not necessary, since you can more easily change “java” to “`apaudit java`” at the point of invocation. However, if you want to enable RootCause in your central Java installation, you should contact OC Systems for assistance: different Java versions are configured differently.

This demonstration program, included as part of the RootCause installation, has been designed to provide an introduction and overview of the RootCause product. The program is:

```
$APROBE/demo/RootCause/Java/Pi.class
```

It is a simple Java program, which computes the value of Pi by iteration using multiple threads. You can find the source in the same directory as the class file.

The goal of this demonstration is to provide an overview of the whole RootCause process, showing initial definition and tuning of the trace, then collection and viewing of more detailed data about a specific function. The demonstration pictured in this chapter was performed on Solaris. Output should be very similar on AIX and Linux.

Set Up

Before running this RootCause Demo, you must install and set up to use RootCause as described in [Chapter 4, "Getting Started"](#). This will define the APROBE environment variable which is necessary to use RootCause.

In the instructions that follow, we'll use \$APROBE to refer to the path where RootCause is installed, for example `/opt/aprobe`.

Use a Supported JRE

Note that only Version 1.2 or higher of the Java runtime environment (JRE) is supported. You can verify this by typing "java -version". If it says something like `java version "1.1.6"` then that will NOT work. See [Chapter 2, "System Requirements"](#) for more information.

Use a Local Disk

We recommend you set your current directory to a disk local to the machine you're running on, though this is not required.

**Defined X-Windows
DISPLAY**

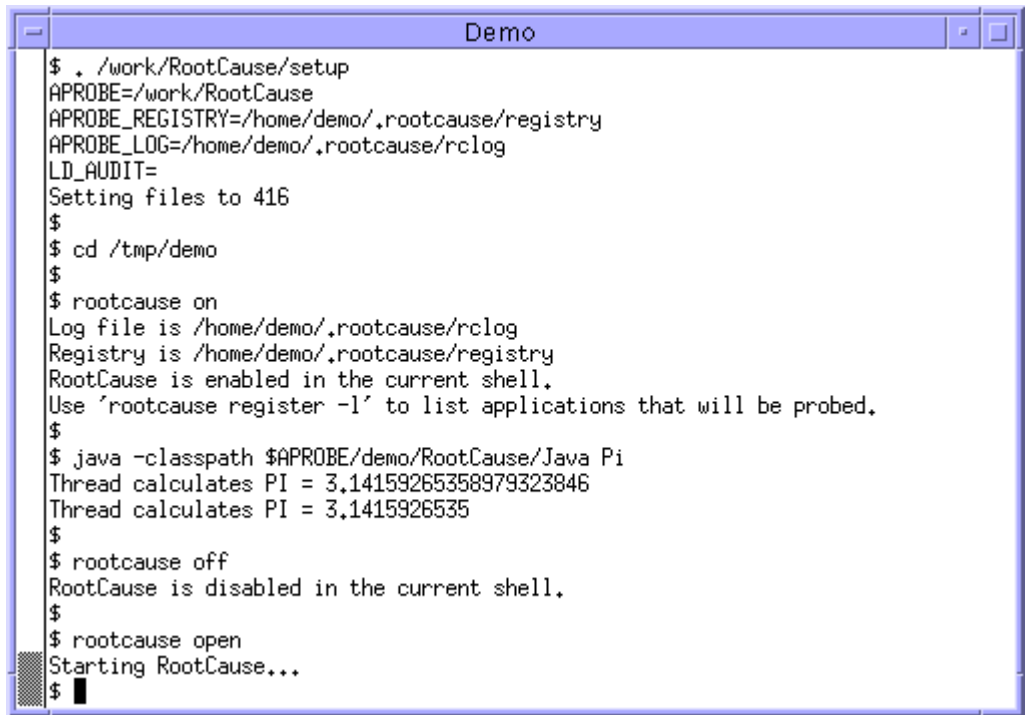
Lastly, make sure your DISPLAY environment variable is set. If you're using a Windows client that is running X emulator software such as eXceed or Reflection, we recommend you move to a Unix display for your initial evaluation. If this is impractical, see "X-emulators: (Exceed, Reflection)" on page 8-47.

Run With RootCause

Run the following commands:

```
rootcause_on
java -classpath $APROBE/demo/RootCause/Java Pi
rootcause_off
```

The `rootcause_on` command enables the automatic logging of every process that is started, and `rootcause_off` disables this logging. When the Pi class is “[registered](#)” with RootCause, it will be traced according to your specifications as well as simply being logged. The illustration below shows the set up and demo execution..



```
Demo
$ ./work/RootCause/setup
APROBE=/work/RootCause
APROBE_REGISTRY=/home/demo/.rootcause/registry
APROBE_LOG=/home/demo/.rootcause/rclog
LD_AUDIT=
Setting files to 416
$
$ cd /tmp/demo
$
$ rootcause on
Log file is /home/demo/.rootcause/rclog
Registry is /home/demo/.rootcause/registry
RootCause is enabled in the current shell.
Use 'rootcause register -l' to list applications that will be probed.
$
$ java -classpath $APROBE/demo/RootCause/Java Pi
Thread calculates PI = 3.14159265358979323846
Thread calculates PI = 3.1415926535
$
$ rootcause off
RootCause is disabled in the current shell.
$
$ rootcause open
Starting RootCause...
$ █
```

NOTE: On AIX, the above process is slightly different than shown above. The invocation of the program must be done with the [rootcause run](#) command directly. For example, the above sequence is changed to:

```
rootcause run java -classpath $APROBE/demo/RootCause/Java Pi
```

See "[Enabling RootCause for an AIX Application](#)" on page 4-4.

View the RootCause Log

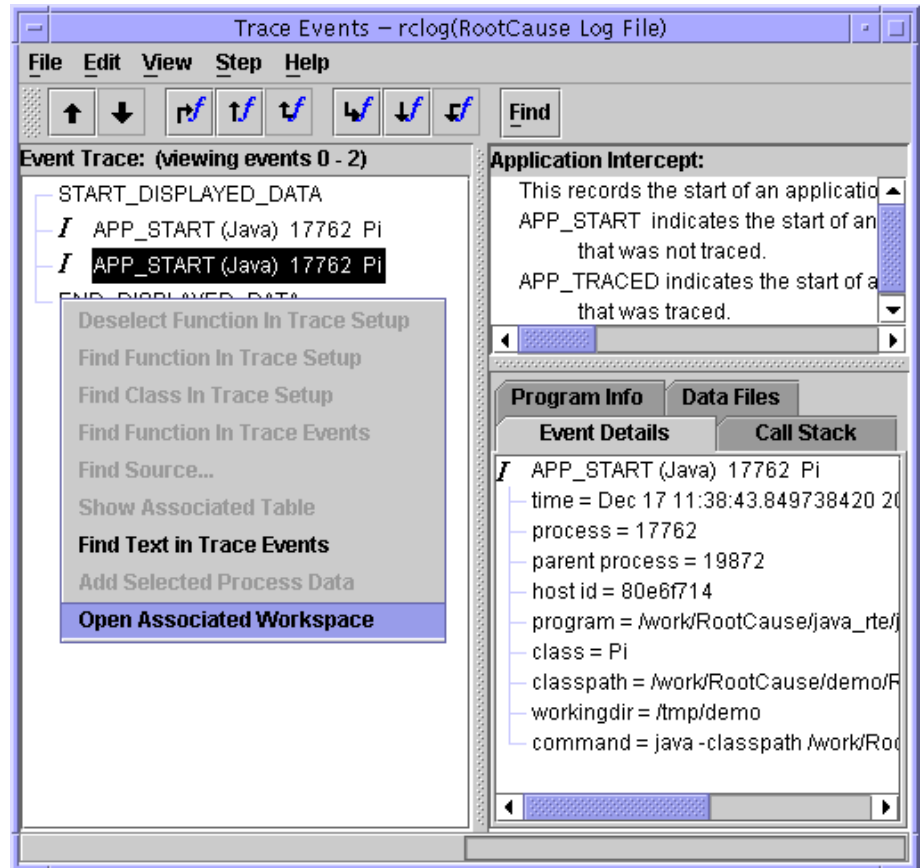
Enter the following command:

```
rootcause open
```

This will open the RootCause main window, and then a view of the RootCause log. This shows the [Trace Display](#) window, the window for viewing all trace events. On the left is the Event Tree; on the upper right is the source/text window; and on the lower right is the “details” window. In the text window you will see some information about the log file.

Locate the APP_START event (in the Trace Event window) associated with the Pi application run earlier. To view information about this event, select the APP_START node in the event tree with a left click. This will fill in “details” about that event in the lower right window. With the APP_START Pi node highlighted, right-click to bring up the [Trace Display Popup Menu](#).

Click [Open Associated Workspace](#) in the popup-menu of the Pi APP_START event.



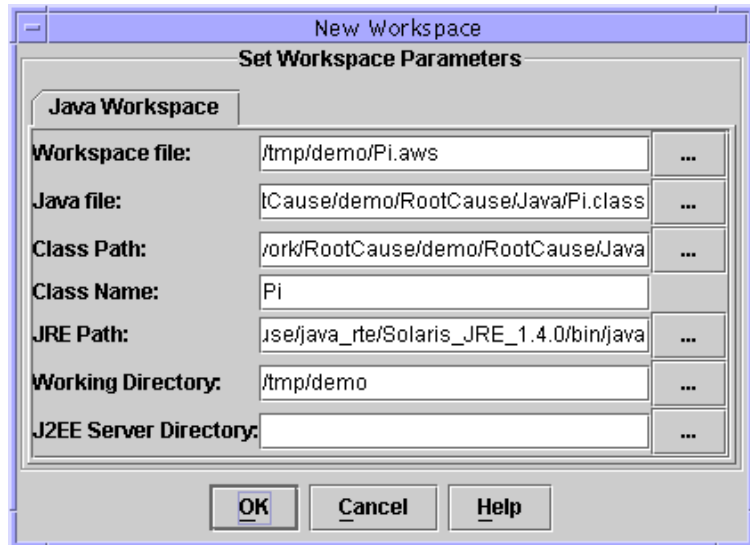
This will open a [New Workspace Dialog](#) with the program name and default workspace filled in.

This combination: selecting a node in the tree, then using the popup menu to choose an operation, is the basic way of working within RootCause.

Create a RootCause Workspace

To complete the creation of a RootCause workspace for the Pi application:

1. Click *Ok* in the *New Workspace dialog* to complete the creation of the workspace.

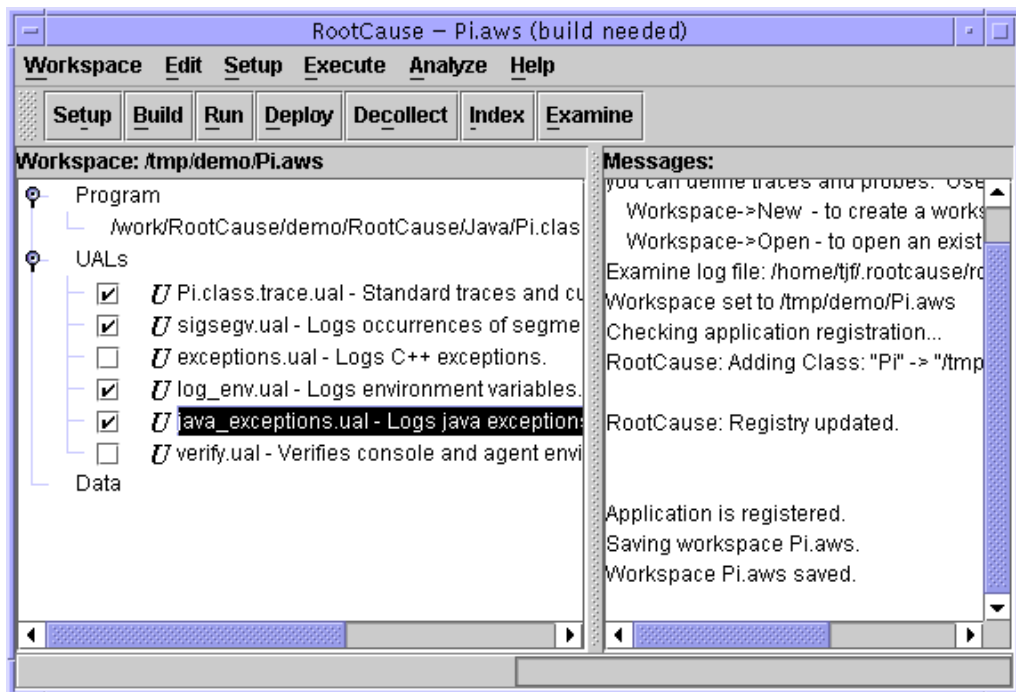


2. Click *Yes* to confirm that you want to register the Pi class with this workspace. This registration is how RootCause determines what applications to trace.



You may close the RootCause Log window opened in a previous step.

You now see the *RootCause Workspace Browser*. This is described in detail in [Chapter 8, "RootCause GUI Reference"](#).



Define the Trace

There are several aspects to a RootCause trace:

- Predefined UALs selected from the [Workspace Tree](#);
- Method and Line traces selected from the [Trace Setup Dialog](#);
- Probes to gather or preserve data, also selected in the [Trace Setup Dialog](#); and
- User-written Custom probes as described in [Chapter 11, "Custom Java Probes"](#).

In this part of the demo we illustrate the use of Predefined UALs and tracing method calls; then we'll return to the Trace Setup Dialog to add some probes. Custom probes are an advanced feature not presented here.

In the RootCause GUI main window, opened in the previous section:

Enable a UAL

Under the [UALs node](#) in the [Workspace Tree](#):

1. Check the checkbox next to predefined UAL labeled `java_exceptions.ual`, as highlighted in the figure above. Simply checking this box will report all user-defined and many predefined Java exceptions.

Define Method Traces

Now we'll add traces specific to this application.

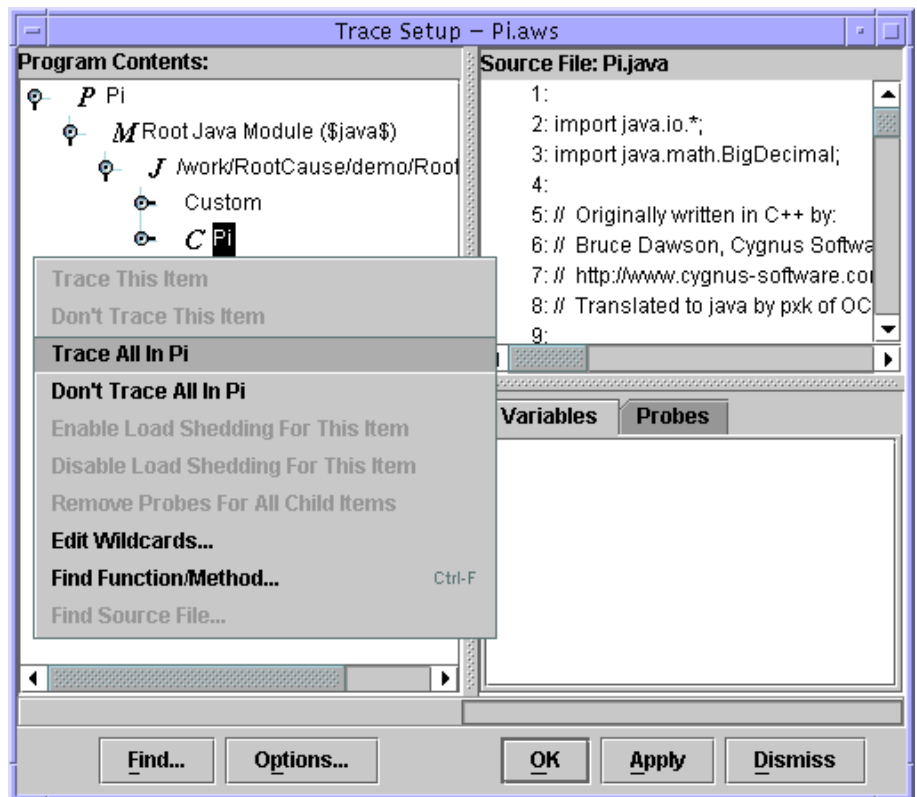
2. Click on the [Setup](#) button in the button bar.

This will open the [Trace Setup Dialog](#), showing the modules of the application in the [Program Contents Tree](#). The [Program Contents Tree](#) identifies the modules, directories, packages, classes and methods in the program, and allows you to specify complex actions on each method. See "[Java Program Contents](#)" on [page 8-22](#) for a description of the Java hierarchy.

For this demo we'll first just specify a trace on all methods in the Pi class, then return later to add data and probes.

3. Click on the "lever" icon next to the **M** Root Java Module node to expand it. You'll see the directories in the class path under this.
4. Similarly, expand the first **J** node, which should be the \$APROBE/demo/RootCause/Java directory from the classpath. You should now see the **C** Pi node, representing the Pi class.

5. Click on the *C Pi* node to select it, then right-click to see the *Trace Setup Pop-up Menu*.
6. Click *Trace All In Pi*.
7. Click the *OK* button at the bottom right of the dialog to record the trace and dismiss the Trace Setup dialog.



Trace With RootCause

As was done in the preceding section, “Run With RootCause” on page 5-3, run the following commands:

```
rootcause_on  
java -classpath $APROBE/demo/RootCause/Java Pi  
rootcause_off
```

Or, for AIX:

```
rootcause run java -classpath $APROBE/demo/RootCause/Java Pi
```

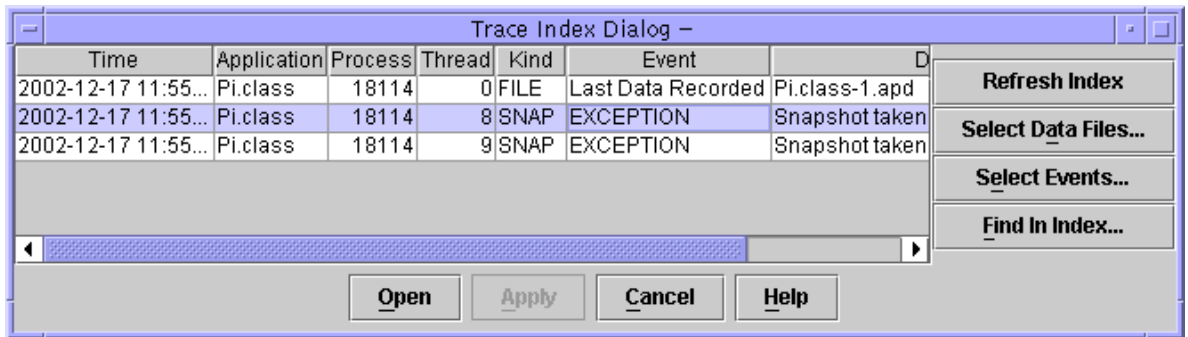
This time, since the program is registered with a workspace, it will be traced as specified in the workspace, and the resulting output will be recorded within the workspace. There will be some startup delay, but if you notice that the program runs slower once started, this is probably because your workspace is being accessed across the network from your machine. See “RootCause and Efficiency Concerns” on page 10-1.

View The Data Index

We're now ready to view the data generated by running with our Trace. This is discussed in detail in "[RootCause Data Management](#)" on page 3-4.

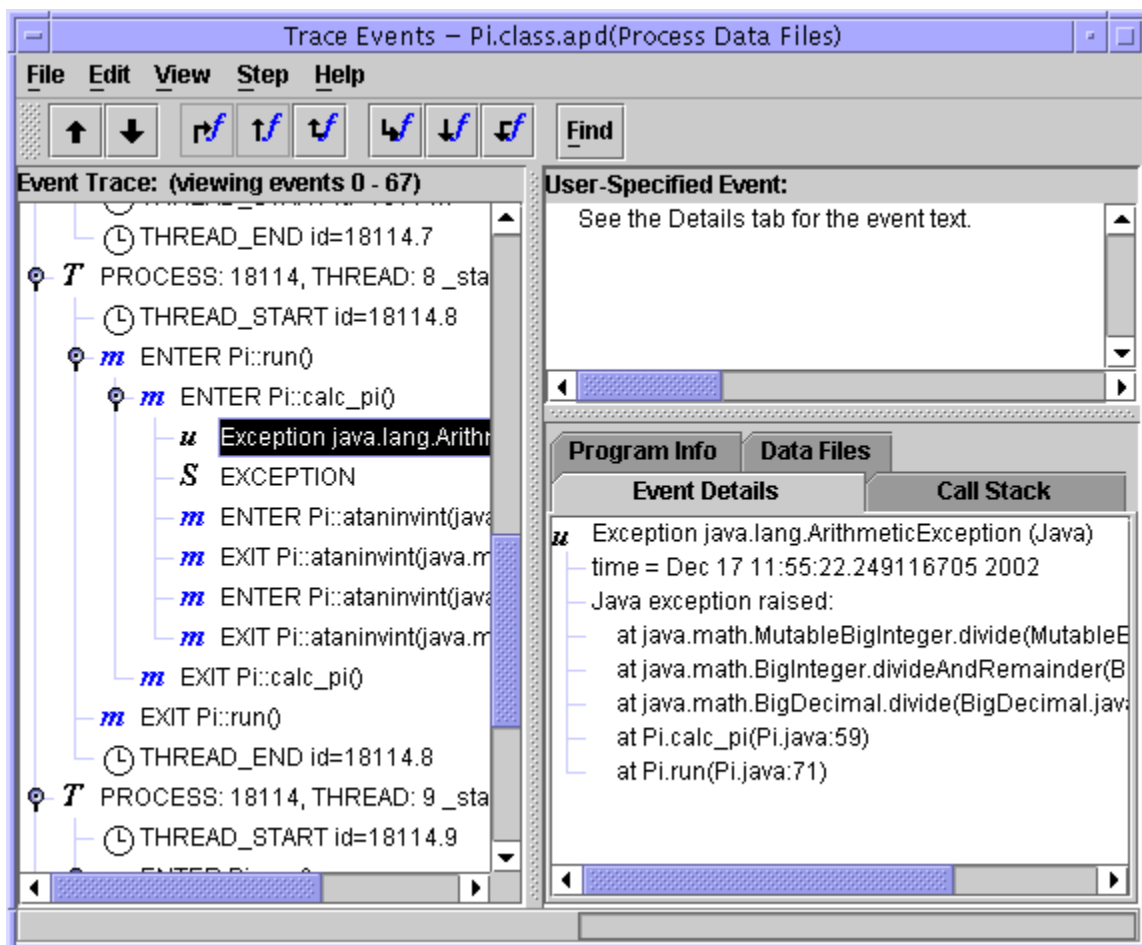
In the [Workspace Browser](#) window, do the following:

8. Click the [Index](#) button. This will bring up the [Trace Index Dialog](#) for the most recently generated data.
9. In the Trace Index Dialog, click the [Select Events](#) button.



10. In the Trace Index Dialog, *double-click* on the second item in the table, the first item with an Event name of EXCEPTION.

This will open a Trace Display Dialog centered at that EXCEPTION snapshot event.



Examine and Revise the Trace

The EXCEPTION trace event selected from the index should appear highlighted in the Trace Display. This was a result of checking `java_exceptions` under ["Enable a UAL"](#) above.

You see that the exception occurred within the method `Pi::calc_pi` from the preceding ENTER node, and see that it occurred before the method `Pi.ataninvint` was entered. These ENTER and EXIT nodes are a result of the “Trace All In Pi” action added under ["Define Method Traces"](#) above.

Use the up-arrow key to go the preceding event labeled “Exception” (in mixed case) to see more information about the exception in the [Event Details Pane](#) in the lower right, as shown above.

If you scroll to the top of the Event Trace, you will see a number of threads. The first is the main thread of the Java application; the last two are those created by the application to compute Pi; the rest are created by the JVM itself and contain no traced calls.

The event tree is a call tree, and can be very useful. From an ENTER or EXIT node in the tree you can use the [Trace Display Popup Menu](#) to:

- remove the called method from the set of methods to be traced,
- find the next reference to the same method in the trace events, or
- go to the method in the [Trace Setup Dialog](#) to trace additional information.

As you step to each event, the [Event Details Pane](#) may show additional information about that event. See ["Trace Display" on page 8-35](#) for a more complete description of this window.

Take some time to explore the event tree. Then we will look at using the information available here to revise or “tune” the trace used in the next run.

Call Counts

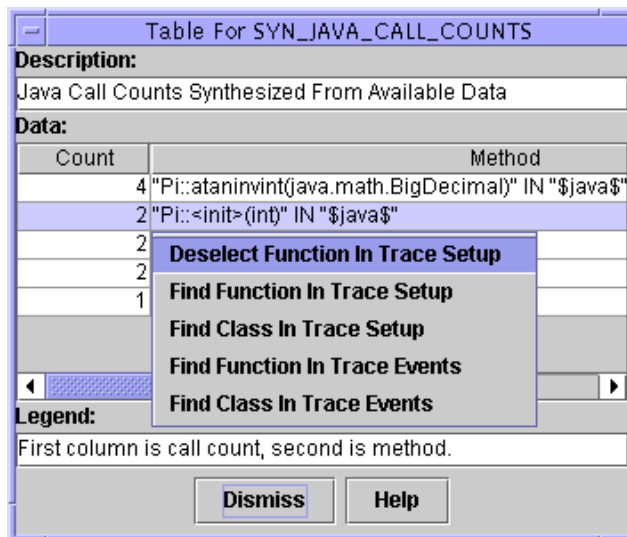
Useful information about the methods called in your program may be obtained by looking at the call frequency as shown in the `CALL_COUNTS` table.

1. Select (left-click) the `SYN_JAVA_CALL_COUNTS` node, near the end of the event tree.
2. Right-click (with MB3) to show the [Trace Display Popup Menu](#) on this node.

3. Click [Show Associated Table](#). This will open a table listing each called method and the number of times it was called.
4. Select (left-click) the "Pi::<init>" entry in the call count table.
5. Right click to show the popup menu.
6. Select [Deselect Function In Trace Setup](#) in the popup menu.

You can also [Find Function In Trace Events](#) to search for methods in the call tree, and remove them from there.

You can also search for methods in the call tree, and remove them from there.



7. When you've finished making changes to the trace, click the *Dismiss* button at the bottom of the SYN_JAVA_CALL_COUNTS table window, and then
8. Click the [Build](#) button in the main window.
9. Notice the effects of removing these calls in the next trace we generate.

NOTE: In most “real” programs, high-overhead functions selected for tracing are automatically identified and disabled via [load shedding](#), and are listed in the [LOAD_SHED Table](#) associated with the LOAD_SHED node at the end of the event tree. This demo doesn’t run long enough for the load shedding heuristics to apply. See ["RootCause Overhead Management" on page 3-7](#) for a general discussion of load shedding.

Tracing The Details

So far we have seen how the RootCause process works by:

- enabling a predefined UAL (`java_exceptions`);
- defining a simple trace from the Trace Setup Dialog;
- running the application under RootCause;
- choosing an event in the Trace Index Dialog;
- viewing events in the Trace Display; and
- modifying the trace by selecting methods from the call counts table.

RootCause allows you to record much more than the entry and exit of methods and threads. You can record data values and insert probes as well.

Open Trace Setup

1. Click the [Setup](#) button in the Workspace Browser to return to the [Trace Setup Dialog](#).

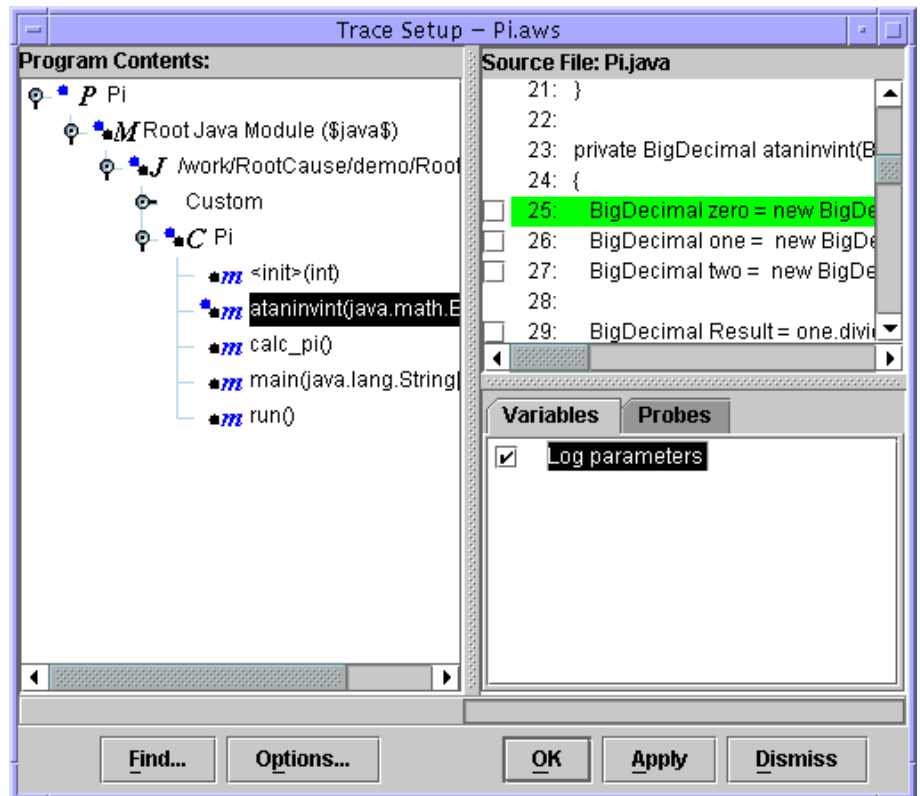
This is just as in “Define the Trace” on page 5-8 above, but this time we’ll record some details about a specific method, `Pi::ataninvint`, rather than tracing everything.

Select A Single Method

2. Expand the Pi class node to see the methods.
3. Select the `ataninvint` method. This will bring up the source code for this method in the [Source Pane](#), and show a Log Parameters checkbox in the [Variables Pane](#).

Log Parameters

4. Check the *Parameters* checkbox.



Add a Snapshot Probe

5. In that same lower-right area, click on the *Probes* tab to show the *Probes Pane*. Probes in this context are special [actions](#) that can be performed at points in the currently selected method.
6. Click the “On” checkbox.
7. Where it says No Trigger, select Function Entry.
8. Where it says No Action, select Log Snapshot.

9. Where it says ROOTCAUSE_SNAPSHOT, select this and type in “My Snapshot” and hit Enter.



We’ve now requested that all parameter values be logged (recorded) on entry to method `Pi::ataninvint`, and also that a data snapshot be taken at this point and marked with the Event Name “My Snapshot”.

Note: A snapshot causes data which might otherwise be deleted do to “data wraparound” to be preserved. In this small demo, a snapshot is not really necessary since it doesn’t generate enough data to wrap around and cause old data to be lost. See ["Data Snapshots" on page 3-6](#) for more information.

Save and Build the Trace

10. Click the *OK* button at the bottom right of the dialog to save and build the trace and dismiss the Trace Setup dialog.

Run With RootCause

11. Again, check that `rootcause` is enabled with either the [rootcause_on](#) or [root-cause status](#) command. Then run the application by running Java on the Pi class, as described in “Trace With RootCause” on page 5-10.

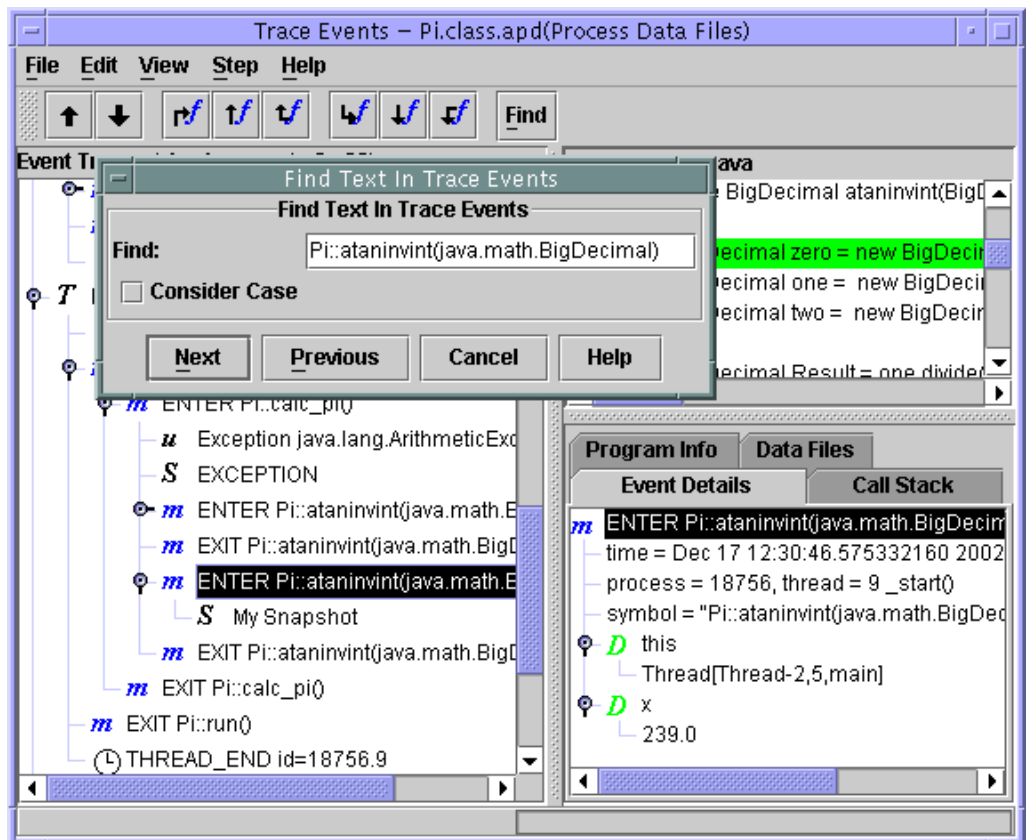
Index the New Trace

12. Click the [Index](#) button in the Workspace Browser window.
13. In the Trace Index Dialog that comes up, double-click on the SNAP entry labeled “My Snapshot” to go right to where our new probes were added.
14. Select the “ENTER ataninvint” node immediately preceding the Snapshot node to see the Parameter values on entry.

Find the Calls Of Interest

The Trace Display window opened from the Trace Index Dialog operation will contain many events, including other calls to `ataninvinvt`.

15. With the “ENTER `ataninvinvt`” node selected, right-click to show the *Trace Display Popup Menu* and choose *Find Function In Trace Events*.
16. This will bring up the *Find Text in Trace Events Dialog* with the current method name filled in. Click Next, and the next occurrence of this string will be selected. Thereafter you can continue clicking *Next*, or enter any other string to search for. You can use *Find Text in Trace Events* from any the pop-up or Edit menu to search for any string in the current Trace Display.



Where To From Here?

This chapter should have given you a good overview of the process of developing a trace and gathering data for a program. Now you're ready to try it on your own application.

Deploying the RootCause Workspace

RootCause performs root cause analysis of problems at the user's site, in the production system. This chapter discusses how to run traces on a remote computer.

Installing The RootCause Agent

RootCause has two components: the RootCause Console and the RootCause Agent. The Agent is the subset of RootCause that is used to run RootCause traces on a remote computer. The Console is the full product that allows one to define and view traces as well as run them (that is, the Console also includes the Agent).

If you're just "trying out" the deploy process on a single computer, your Console installation can also serve as the remote installation. If you really are deploying RootCause to a remote site you will want to install just the RootCause Agent subset there, and enable remote execution using RootCause agent licenses.

Follow the installation directions in "RootCause Agent Installation" on page 2-7 to install the RootCause Agent on the remote computer; this only needs to be done once per remote computer.

After this is installed on the remote computer, you deploy RootCause trace definitions to the remote RootCause Agent and get back files that contain the logged trace data.

The Agent installation does not have its own license—the license is delivered with the deployed probes. Instead, you will obtain one or more agent license keys from OC Systems and append them to the file

```
$APROBE/licenses/agent_license.dat
```

in your RootCause Console installation. Agent licenses will be automatically copied into the deployed workspace (see “Deploying A RootCause Workspace” on page 6-2). See “Licensing” on page 2-8 for a more in-depth discussion on licensing issues.

Building a “Traceable” Application

Any Java application is traceable with RootCause so long as it’s running under a supported JRE. See ["System Requirements" on page 2-2](#), and “Your Application and Different JREs” on page 10-12

Deploying A RootCause Workspace

When you have built and tested the traces and probes, and you want to apply them to an application that exists at (or will be shipped to) a remote site, you are ready to deploy the workspace containing those traces and probes. This workspace will be your “flight recorder” for the application at the remote site. To generate the deploy (.dply) file:

1. Confirm that you’re developing and testing with the same build of the application you’ll be shipping. See “Building a “Traceable” Application” on page 6-2.
2. Develop and test your traces locally. When you see the information you think you’ll need at the remote site, then you’re ready to deploy.
3. Enable the [verify](#) predefined UAL in the [Workspace Browser](#) window. This checks the correspondence between the program and modules on the remote system with those in the local (formatting) environment, and alerts the user when an incompatibility is detected.
4. Click on the [Deploy](#) button in the main [Workspace Browser](#) window. This will display the [Deploy Dialog](#).
5. In the Deploy Dialog, enter a file name for your deploy file. This file will be created by RootCause, and it will contain the trace definitions for your current workspace.

6. In the Deploy window, click “OK”. RootCause will attempt to create the `dply` file. For example, if you’re using the workspace created in [Chapter 5, “RootCause Demo”](#), then this will create `dply`.

The `.dply` file will also contain the license needed to run RootCause on the remote computer. When the `.dply` file is created, the file `$APROBE/licenses/agent_license.dat` mentioned in see “Installing The RootCause Agent” on page 6-1 is automatically included. If the license file is not found, or does not contain a license, an error dialog will appear. If you see this, you can click “Yes” to proceed with deploying, or “No” if you wish to investigate getting a valid Agent license.

7. Transfer the `.dply` file over to the remote computer (be sure to specify binary mode if you use ftp).

Registering a Deployed Workspace

8. On the [remote](#) computer open an xterm or other command shell.
9. Register the application on the remote computer using the [rootcause register](#) command. For example:

```
rootcause register Pi.dply
```

This creates a workspace in the current directory (or a path specified with the `-w` option) and registers the workspace with the main Java class.

10. Enable RootCause on the remote computer in the environment where the application will be run, using the [rootcause_on](#) command. Note that this needs to be done in a parent process of the shell that will run the application(s) so that the application(s) will inherit the value of the environment variables it sets.

Collecting Data At The Remote Site

11. Run the application(s) as you normally would.
12. When you want to [collect](#) and examine the RootCause trace data, execute the [rootcause collect](#) command on the remote computer. For example:

```
rootcause collect -c Pi
```

Many classes may be collected at once by listing them on the collect command line. The applicable files in each registered workspace will be compressed into a single `.clct` file.

Formatting and Viewing the Remotely-Collected Data

13. Transfer the `.clct` file back to the local computer where the RootCause [Console](#) is installed (be sure to specify binary mode if you use ftp).
14. In the RootCause GUI, click on the [Decollect](#) button in the main window. This will display the [Decollect Data Dialog](#). If the RootCause GUI is not currently running, you may open the collect file when starting the GUI, for example:

```
rootcause open file.clct
```
15. In the [Decollect Data Dialog](#), enter the name of the `.clct` file and the destination directory into which the file will be expanded. The destination directory should be empty as the [decollect](#) operation will expand a number of files into this directory. Then click “OK”.

This will create a [decollection](#), a directory, with suffix `.dcclct`, containing the workspaces collected from the remote computer. It will then proceed with the [Open Decollection](#) operation, which will show a [Trace Index Dialog](#) for the newest data in the decollection.
16. Select the event(s) you wish to view, or use [Select Data Files](#) to change the data that was indexed.
17. Format the data into a Trace Display. If you used the [verify](#) predefined UAL as suggested in "[Deploying A RootCause Workspace](#)" on page 6-2, you'll see a TEXT node identifying any mismatches that may cause formatting problems.
18. From the Trace Display for the decollected data, you can use [Add Data Files To Display](#) to add in the Decollected RootCause Log file and other data.

RootCause Files and Environment Variables

RootCause consists of a [GUI](#) to define traces, the Aprobe runtime to implement those traces, and a number of directories, files, and variables in the execution environment which control and record when the traces are applied. This chapter briefly describes those files and environment variables.

Workspace

All activity is performed in the context of a workspace. A workspace is a directory which contains all the information about the target program configuration, the configuration of the trace setup, and the resulting log files.

The contents of the workspace are manipulated by the RootCause GUI. If you change something within the workspace, such as a script or configuration file, the changes may be lost the next time a rootcause GUI operation is performed.

UAL File

A user action library ([UAL](#)) consists of a set of probes that are attached to the target program when it runs. The probes are activated when specified portions of the target program are reached. Some [predefined UALs](#) may be selected in the RootCause GUI [Workspace Tree](#), additional ones are provided in the underlying Aprobe product, and the user can build custom UALs. As a user of RootCause, you do not need to be concerned with these files. RootCause handles them automatically. If you do use the underlying power of Aprobe, then you can create your own probes and your own UAL files.

XMJ File

An XMJ file, also known as a *Java Probe deployment descriptor file*, is a user-created text file containing an XML description of Java probes. The DTD for this is on-line in \$APROBE/html/xmj.dtd.

Data (APD) File

Aprobe places the [logged](#) data from a traced [executable](#) into a [Data File](#), also called an [APD file](#). For the most part, RootCause users are isolated from this; however, when formatting the data, you can choose the data files you wish to view or generate an index from. See "[Bounding Total Data](#)" on page 3-5 for more information.

Process Data Set

Process Data Sets are used to collect more than one wrap-around set (or ring) of data files for a single program. See "[Data for Multiple Processes](#)" on page 3-5 for more information.

Deploy File

A [deploy](#) file (with extension `.dply`) is created by the [Deploy](#) operation in the RootCause GUI. It contains all the information needed by RootCause to trace a (possibly [stripped](#)) program at a [remote](#) site. The deploy file is transmitted to the remote computer and installed there to enable tracing of the program on the remote computer.

Collect File

A [collect](#) file (extension `.clct`) is created by the [rootcause collect](#) command. It contains all the information collected by RootCause for a traced program at a remote site. The collect file is transmitted from the remote computer to the local computer where it is "[decollected](#)" to examine its contents.

Decollection

The [Decollect](#) button in the RootCause GUI unpacks a `.clct` file created by the [rootcause collect](#) command. This creates a directory containing all the decollected workspaces. This directory is known as a Decollection (with extension `.dclct`), and is directly accessible using the [Open Decollection](#) and [Recent Decollections](#) items in the RootCause GUI's [Workspace Menu](#).

RootCause Registry

To use RootCause on an application, you must first [register](#) the application by adding it to your RootCause [registry](#). The RootCause GUI will do this automatically, and there is a command line interface as well (see the [rootcause register](#) command).

RootCause will only trace applications defined in the RootCause Registry.

On any one computer, there may be one or possibly many different registries, depending on your desired use. For example, it would be common for each user to have his/her own registry. Also, there may be one registry for the whole computer if it is a dedicated server and there is an integrated set of RootCause probes designed for that server.

The location of the RootCause registry is defined by the [APROBE_REGISTRY](#) environment variable, and is generally under the [.rootcause Directory](#) so that each user has a separate registry.

All manipulations of the registry are done using the [rootcause register](#) command, or via the items in the RootCause GUI's Workspace menu.

Important: The registry is meant to be manipulated only with the `rootcause register` command (See “rootcause register” on page 9-17). Do not change its contents by any other means!

RootCause Log

The RootCause log file records what programs are started and what programs are traced while RootCause is enabled. Programs that are started but not traced are recorded as APP_START events. Programs that are traced by RootCause are recorded as APP_TRACED events. The RootCause log file may also contain other messages and debug information as TEXT events. The RootCause log file may be examined in the GUI and used as a starting point for creating or opening the workspaces associated with programs recorded in the log file.

By default RootCause writes a line to the log file whenever an application is [run with rootcause on](#), or fails to run due to an error. If you wish to record only those applications that are traced, you may change the verbosity level with the [rootcause register](#) command.

To log only applications that are registered and probed:

```
rootcause register -s verbose -e off
```

To log all applications that are executed when rootcause is enabled, whether or not they are registered:

```
rootcause register -s verbose -e on
```

While becoming familiar with RootCause, you may want to examine the log file often. You can display it using the [rootcause log](#) command.

.rootcause Directory

The directory

`$HOME/.rootcause`

is created and used by the RootCause GUI to maintain some user-specific attributes of your RootCause environment:

- the [RootCause Log](#) file is located here by default
- the [RootCause Registry](#) is located here by default as well.
- a customized [rootcause.properties](#) file may be placed here.
- a `preferences` file in this directory contains the recent workspaces and other RootCause GUI internal settings.
- temporary files are created here.

The directory is called `.rootcause_linux` on Linux and `.rootcause_aix` on AIX to avoid collisions and confusion on mixed systems sharing a common `$HOME` directory.

The [APROBE_HOME](#) may be used to specify a different location.

rootcause.properties

The file

`$APROBE/lib/rootcause.properties`

defines some properties used to determine the RootCause appearance. In general, you need not be concerned with these, but if you wish to change the background color or other property defined by `UIManager` you can do so by placing an edited version of this file in the [.rootcause Directory](#).

setup Script

The files `$APROBE/setup` and `$APROBE/setup.csh` are provided to define the RootCause environment in your shell. See [Chapter 4, "The Setup Script"](#) for details.

Environment Variables

APROBE	The APROBE environment variable points to the RootCause product installation directory, and is automatically defined by the setup script. We suggest that you do not modify this environment variable.
APROBE_HOME	<p>The APROBE_HOME environment variable defines a non-default location for the .rootcause Directory.</p> <p>If APROBE_HOME is defined when the \$APROBE/setup or setup.csh script is run, the RootCause Log and RootCause Registry files are created under \$APROBE_HOME. In this way, one can have a single system-wide RootCause environment by setting APROBE_HOME globally, and creating world-accessible registry and log files there.</p>
APROBE_JRE	The APROBE_JRE environment variable specifies a non-default JRE (Java installation) to use instead of the one shipped with RootCause. See " Platform-Specific GUI Issues " on page 8-47.
APROBE_JAVA_HEAPSIZE	The APROBE_JAVA_HEAPSIZE environment variable specifies a non-default Java heap sized to be use by the RootCause console GUI. The value of this environment variable is the entire Java parameter value. The default is "-Xmx128m".
APROBE_LOG	The APROBE_LOG environment variable specifies the location of the RootCause Log file (see " RootCause Log " on page 7-3). It is initialized by the setup script (see " The Setup Script " on page 4-1) to the name rclog under the .rootcause Directory . If unset, the location \$APROBE/arca/rc.log is used. Generally the RootCause Log and RootCause Registry are kept in the same directory, defined by the value of the APROBE_HOME environment variable, so you shouldn't have to set this directly.

APROBE_REGISTRY

The APROBE_REGISTRY environment variable specifies the location of the RootCause registry file (see "[RootCause Registry](#)" on page 7-2). It is initialized by the [setup Script](#) to the name `registry` under the [.rootcause Directory](#). If unset, the location `$APROBE/arca/registry` is used. Generally the [RootCause Log](#) and [RootCause Registry](#) are kept in the same directory, defined by the value of the [APROBE_HOME](#) environment variable, so you shouldn't have to set this directly.

APROBE_SEARCH_PATH

The APROBE_SEARCH_PATH environment variable identifies directories to be searched by RootCause for source files when the file is not found in the directory from which it was compiled. If defined, The APROBE_SEARCH_PATH environment variable is a colon-separated list of directories (like `PATH` and `LIBPATH`) in which to search for a source file to display, if the file is not found in the directory recorded in the class file.

For example, if a class is compiled from files in directories `/build/common`, `/build/console`, and `/build/gui`, and the directory `/build` has been moved to `/old/build`, you could do:

```
export APROBE_SEARCH_PATH=\
    "/old/build/common:/old/build/console:/old/build/gui"
```

Use of the environment variable to find source files is not usually necessary, since the RootCause GUI provides an interface for specifying the search path the first time it's needed, and this path is recorded in the workspace.

The APROBE_SEARCH_PATH environment variable is also used by RootCause and Aprobe to find *object* files that have been moved -- see Appendix A of the Aprobe User's Guide.

LD_AUDIT (Solaris)

The LD_AUDIT environment variable is recognized by the **Solaris** operating system and is used by RootCause to hook into applications that are registered by RootCause. The commands `rootcause_on` and `rootcause_off` will set and unset the LD_AUDIT environment variable for RootCause.

The RootCause setup script does not set the LD_AUDIT environment variable, so you will need to execute `rootcause_on` after the setup script to actually cause RootCause to start examining each new process for probing. LD_AUDIT

is a normal Solaris environment variable with the usual semantics about being inherited by sub-processes, etc.

At OC Systems, we have set LD_AUDIT at system boot time, so all processes will be examined by RootCause as they launch, to ensure its robustness and low overhead. But you can limit the scope of RootCause by limiting the scope of where the LD_AUDIT environment variable is set, even though the overhead imposed by this checking is small.

For Solaris 7 and higher, the LD_AUDIT_64 environment variable may also be set to point at a dummy 64-bit library so that the runtime linker does not issue warning messages. This library does not invoke RootCause because 64-bit applications are currently not supported.

Note that the LD_PRELOAD environment variable is *not* used by RootCause on Solaris.

LD_PRELOAD (Linux)

The LD_PRELOAD environment variable is recognized by the **Linux** operating system and is used by RootCause to hook into applications that are registered by RootCause. The commands `rootcause_on` and `rootcause_off` will set and unset the LD_PRELOAD environment variable for RootCause.

The RootCause setup script does not set the LD_PRELOAD environment variable, so you will need to execute `rootcause_on` after the setup script to actually cause RootCause to start examining each new process for probing.

LD_PRELOAD is a normal Linux environment variable with the usual semantics about being inherited by sub-processes, etc. You can limit the scope of RootCause by limiting the scope of where the LD_PRELOAD environment variable is set, even though the overhead imposed by this checking is small.

Note that LD_PRELOAD may be used by other tools or even your own application. In such cases you must take care in updating this variable: contact OC Systems for assistance in this case.

For Solaris 7 and higher, the LD_AUDIT_64 environment variable may also be set to point at a dummy 64-bit library so that the runtime linker does not issue warning messages. This library does not invoke RootCause because 64-bit applications are currently not supported.

AP_ROOTCAUSE_ENABLED (AIX)

AIX has no mechanism corresponding to LD_AUDIT or LD_PRELOAD that allows libraries to be specified at load time, and hence one cannot just set an environment variable to start intercepting processes on AIX. Instead, one identifies a program that one may want to intercept, renames the program executable, and replaces it with a script called `run_with_apaudit`, as described in ["Enabling RootCause for an AIX Application" on page 4-4](#). This script recognizes the AP_ROOTCAUSE_ENABLED environment variable, which is defined by the `rootcause_on` alias and unset by the `rootcause_off` alias.

RC_WORKSPACE_LOC

When an application is [run with rootcause on](#) and is [registered](#) with a [workspace](#), the location of that workspace is defined in the environment variable RC_WORKSPACE_LOC prior to the application being run with `aprobe`. This allows one to use this environment variable to reference files in `aprobe` options or within custom probes. This is especially useful in specifying the location of the configuration file needed by a user-defined UAL in the [Aprobe Parameters](#) field of the [Add UAL Dialog](#). This environment variable is also defined when data is formatted, and so can be used in [Apformat Parameters](#) as well.

RC_SHORT_WORKSPACE_LOC

If the path to your workspace may contain blanks (such as is common on Windows platforms), you should use the RC_SHORT_WORKSPACE_LOC environment variable instead of the [RC_WORKSPACE_LOC](#).

RootCause GUI Reference

This chapter describes all parts of the Rootcause Console Graphical User Interface, or the “RootCause GUI” for short. It is meant to serve as a reference to supplement the text provided by the Help buttons on the windows themselves.

Workspace Browser

The *Workspace Browser* is the main window of the RootCause GUI. The Workspace Browser controls the program tracing process.

The Workspace Browser is composed of the following parts:

- the [Workspace Tree](#) on the left side;
- the [Message Pane](#) on the right side;
- the menu bar across the top; and
- the [Toolbar](#) below the menu bar.

Workspace Tree

The *Workspace Tree* displays information about the current workspace. There are three sections to this tree: [Program node](#), [UALs node](#), and [Data node](#). A [Workspace Tree Popup Menu](#) is provided to perform operations on nodes in the Workspace Tree.

Workspace Tree Popup Menu

Select a node in the Workspace Tree and use the right mouse button (MB3) to display a popup menu. The following operations from the Workspace and Setup menus are provided, depending on the node selected:

- [Workspace Menu: Reset Dynamic Module](#)
- [Workspace Menu: Delete Dynamic Module](#)
- [Setup Menu: Edit UAL](#)
- [Setup Menu: Remove UAL](#)

Program node

The program configuration is displayed under the Program node. This includes the main program and any dynamic modules used by the program as specified by the user using [Add Dynamic Module](#) in the [Workspace Menu](#).

UALs node

The configuration of user action libraries (UALs) is displayed under the UALs node. The trace predefined UAL is always present, and other predefined UALs are displayed depending upon the local configuration. Each can be selected by checking the box to the left of the UAL name, and disabled by clearing the checkbox.

NOTE: a [Build](#) operation is required to apply the changes made to the UALs tree.

The predefined UALs are program-wide actions that can be optionally applied by clicking the checkbox next to the name. Additional predefined UALs may be written and added with [Setup->Add UAL](#), or by directly editing the file `$APROBE/ual_lib/predefined_uals`.

log_env: Use the *log_env* predefined UAL to collect information about the environment in which the program is running. This information includes environment variables, the current user and machine, and other information. This information will appear in the [Program Information Pane](#) of the [Trace Display](#) window.

exceptions: Enable the *exceptions* predefined UAL to trace C++ (and Ada) exceptions that occur in the program. These will show up as exception events in the [Trace Index Dialog](#), and a full [traceback](#) will appear in the [Event Details Pane](#) of the Trace Display window.

java_exceptions: Enable the *java_exceptions* predefined UAL to trace Java exceptions that occur in the program. These will show up as exception events in the *Trace Index Dialog*, and a full *traceback* will appear in the *Event Details Pane* of the Trace Display window. In addition, some Java run-time exceptions will cause a *snapshot* to be taken as well. The actions associated with specific Java exceptions may be specified using the *Java Exceptions Configuration Dialog*.

verify: Enable the *verify* predefined UAL to verify the traced modules are the same between run-time and format-time. This introduces some startup overhead, but is recommended when deploying a workspace for use in a different environment. See "*Deploying A RootCause Workspace*" on page 6-2.

sigsegv: The *sigsegv* predefined UAL, enabled by default, logs a traceback when one of several program-termination signals occurs: SIGQUIT, SIGILL, SIGABRT, SIGBUS, SIGSEGV, or SIGTERM. The traceback will appear in the *Event Details Pane* of the Trace Display window.

Data node

The most recent *Process Data Sets* are shown under the *Data node* in the *Workspace Tree*, most-recent-first. Double-clicking on a PROCESS DATA node will open and update the *Trace Index Dialog* for that data.

Data is recorded when the program registered with the current workspace is *run with rootcause on*. The data is organized per-process into a *Process Data Set*, and is identified by the process id. By default, only the two most recent data sets are kept; older ones are deleted. More data may be preserved by increasing the value labeled "*Keep logged data for N previous processes*" in the *RootCause Options Tab* of Options Dialog opened from the *Setup Menu*.

Message Pane

The *Message Pane* displays information about the operations performed on the workspace. A popup menu is accessible (via the right mouse button) in the message pane, and provides access to the operations in the *Edit Menu*.

Workspace Menu

The *Workspace Menu* is the leftmost menu in the Workspace Browser window and contains many fundamental operations.

New: You must have a [workspace](#) to begin work tracing a program. A single program is associated with each workspace. You can create a new workspace using *New*. First the current workspace will be saved, if it has been changed but not yet saved. The [New Workspace Dialog](#) allows you to set the name and location of the workspace and set the program associated with the workspace.

Open: The user can work with an existing workspace by choosing *Open*. First the current workspace will be saved, if it has been changed but not yet saved. Then the user can choose an existing workspace and it will be loaded and the user can resume from where the workspace was last saved.

Save: The current workspace can be saved, if it has been modified, by choosing *Save*. All information about the program configuration and traces will be saved.

Save As: The current workspace can be renamed by choosing *Save As*. The user will choose the new name of the workspace, and it will be saved to the workspace file. The new workspace *must be rebuilt* using the [Build](#) operation before it can be accessed by a running program. If a workspace file with the chosen name already exists, the user will be asked if the file should be overwritten. The user can cancel the operation or proceed to overwrite the existing workspace file.

Close: Choose *Close* to close the workspace without exiting RootCause. You will be asked if you want to save the workspace, if it is needed.

Recent Workspaces: The most recently visited workspaces can be opened by choosing them from the *Recent Workspaces* submenu. The chosen workspace will be loaded and work can resume from where the workspace was last saved.

Open RootCause Log: The [RootCause Log](#) can be examined by choosing *Open RootCause Log*. The RootCause log records which programs have been started and which have been traced since RootCause was enabled. This can be a good starting point for determining which programs should be traced in a large, multi-program application suite.

Decollect Collected: Choose *Decollect Collected* to unpack the data collected on a remote computer (after a deploy operation). The decollected data will consist of one or more workspaces of data. This will open the [Decollect Data Dialog](#) to choose the source (.clct) file and destination (.dclct) directory. Upon successful completion, a [Trace Index Dialog](#) is opened to index the newest data within the [decollection](#).

Open Decollection: Decollected workspaces, which have been collected from a remote computer, can be examined using *Open Decollection*. The user will choose a [decollection](#), which was produced by a previous *Decollect Collected* operation, and a *Trace Index Dialog* will be opened.

Recent Decollections: The most recently visited decollections can be opened by choosing them from the *Recent Decollections* submenu. A *Trace Index Dialog* is opened to index the newest data within the [decollection](#).

List RootCause Registry: The RootCause registry defines what workspaces apply to which applications. You can list the current contents of the RootCause registry with *List RootCause Registry*.

Register Program: The current workspace can be [registered](#) with its corresponding program or class using *Register Program*.

Unregister Program: You can remove the RootCause [registry](#) entry for the current workspace/program with *Unregister Program*.

Reset Program: You can reset the program associated with a workspace if it has been rebuilt, or its location has changed. Choose *Reset Program* to select the new version of the program. When the program is reset, the existing trace setup will be checked against the new version, and any invalid traces will be discarded. Traces are invalid if the method/data no longer exists in the program.

Add Dynamic Module: Any number of runtime-loaded [dynamic modules](#) can be added to the program configuration by choosing *Add Dynamic Module*. The user will then select the dynamic module through a file chooser dialog. If this module is not already part of the program configuration it will be added to the program configuration and displayed in the workspace tree. You can choose either a class file or JAR file to be the program. You can add other class files or JAR files as dynamic modules. If you have a C++ license also, you can also add shared libraries to trace JNI calls as well.

Reset Dynamic Module: You can reset a dynamic module associated with a workspace if it has been rebuilt, or its location has changed. Select a dynamic module in the workspace tree then choose *Reset Dynamic Module* to select the new version of the module. When a module is reset, the existing trace setup will be checked against the new version, and any invalid traces will be discarded. Traces are invalid if the method/data no longer exists in the module.

Delete Dynamic Module: You can remove a dynamic module associated with a workspace by selecting a dynamic module in the workspace tree and choosing *Delete Dynamic Module*. When a module is deleted any traces within the module will be lost.

Update J2EE Modules: You can add or change the J2EE modules associated with a workspace with *Update J2EE Modules*. This pops up a File Chooser dialog with which you enter the directory where deployable Enterprise Java Bean (EJB) and Servlet classes and jars reside. If you've already specified a path, this will cause that to be re-read and the list of Java modules updated. See "[RootCause J2EE Support](#)" on page 10-14 for more information.

Exit: Choose *Exit* to terminate the RootCause GUI. You will be asked if you want to save the workspace, if it is needed.

Edit Menu

The *Edit Menu* supports Copy/Cut/Paste/Delete from the Message pane. If you are using RootCause on a version of Solaris less than 8, the Copy operation here is the only way to copy text from the Message window. If you are using Solaris 8, the normal mouse copy/paste operations work directly. These operations are also available on a popup (context) menu by right-clicking within the Messages window.

Cut: Choose *Cut* to cut the selected text from the message pane and copy it to the system clipboard.

Copy: Choose *Copy* to copy the selected text from the message pane to the system clipboard.

Paste: Choose *Paste* to paste text from the system clipboard into the message pane at the current cursor position.

Delete: Choose *Delete* to delete the selected text from the message pane.

Setup Menu

Once you have chosen a workspace and defined the program configuration, you can use the items in the *Setup Menu* to set up options.

Trace UAL: Choose *Trace UAL* to set up options to control the predefined trace UAL. This is the primary means for tracing program execution. This will open the [Trace Setup Dialog](#).

Add UAL: Choose *Add UAL* to add a new UAL to the workspace. This allows you to add your own “predefined” UALs to a workspace and control their configuration. This will open the *Add UAL Dialog*.

Edit UAL: Choose *Edit UAL* to edit parameters or other configuration associated with the selected UAL. (This is insensitive unless a UAL is selected.) This will open the UAL-specific configuration dialog, such as the *Trace Setup Dialog* or *Java Exceptions Configuration Dialog*; or else the default dialog which simply allows UAL parameters to be specified when the UAL is used by aprobe and aformat. Note that the “-p” argument that is used to indicate parameters for a UAL should *not* be specified here.

Remove UAL: Choose *Remove UAL* to remove a UAL from the list displayed under the **UALs node**. This item is insensitive if no UAL is selected, or if the selected UAL is predefined.

Build: Choose *Build* to compile the trace setup into a UAL in preparation for running the program locally or deploying the trace to a remote computer.

Options: Choose *Options* to open the *RootCause Options Dialog*, from which most workspace configuration items are selected.

Source Path: Choose *Source Path* to set up the search path used to find source files in the Trace Setup and Trace Display windows. Source files will be displayed when you define a trace, and also when you view logged trace events. This will open the *Edit Source Path Dialog*.

Class Path: Choose *Class Path* to set up the class path for Java program. This will define where the Java Virtual Machine (JVM) searches for class and JAR files. The class path is used only for Java programs. This will open the *Edit Class Path Dialog*.

JRE Path: Choose *JRE Path* to set up the path to the Java Runtime Environment. This determines which Java Virtual Machine will be used to run your Java program when run from the Execute menu within RootCause. The JRE path is used only for Java programs. This will open the *Java Path Dialog*.

Execute Menu

Once you have set up the traces you want for your program you can use items in the *Execute Menu* to define other aprobe options, and run the program with the traces.

Run Program: Choose *Run Program* to run the program on the [local](#) computer (the same computer running the RootCause GUI) with the defined traces. This option is useful when developing traces on the local computer. Registered programs can be run outside of RootCause from the command-line or from a script or batch file, but this may be more convenient. This will open the [Run Program Dialog](#).

Deploy Program : Choose *Deploy Program* to deploy the current set of traces to a [remote](#) computer where the RootCause Agent is installed. The traces and options selected in the current workspace will be packaged up to be sent to the remote computer. This will open the [Deploy Dialog](#).

Analyze Menu

Once you have run your program with the traces you specified, you can analyze the logged data.

Index Process Data: Choose *Index Process Data* to build an index for the most recently logged data in the workspace. This will open the [Trace Index Dialog](#).

Examine Process Data: Choose *Examine Process Data* to examine the most recently logged data in the workspace. This will open the [Trace Data Dialog](#).

Help Menu

Use the *Help Menu* to figure out what is going on and how to get things done.

Toolbar

Use the *Toolbar* to access common menu items quickly.

Setup: Choose *Setup* to set up traces. This will open the [Trace Setup Dialog](#).

Build: Choose *Build* to compile the trace setup in preparation for running the program locally or deploying the traces to a remote computer.

Run: Choose *Run* to run the program locally. This will open the [Run Program Dialog](#). Registered programs can be run outside of RootCause from the command-line or from a script or batch file, but this may be more convenient.

Deploy: Choose *Deploy* to deploy the traces to a remote computer. This will open the [Deploy Dialog](#).

Decollect: Choose *Decollect* to unpack data collected from a remote computer. This will open the [Decollect Data Dialog](#).

Index: Choose *Index* to build an index for the most recently logged data in the workspace. This will open the [Trace Index Dialog](#).

Examine: Choose *Examine* to examine the most recently logged data in the workspace. This will open the [Trace Data Dialog](#).

New Workspace Dialog

The *New Workspace Dialog* permits the user to create a new [workspace](#). A single Java file is associated with each workspace. If you have a license for both Java and C++, two tabs will be visible; select the tab corresponding to the kind of workspace needed. This user's guide describes only the Java workspace.

Java Workspace

The Java file name can be entered in the text field labeled Java File or can be selected using the “...” button to the right of the text field. The Java file can be either a Java `.class` or `.jar` file. If a `.jar` file is selected, the main class name must be specified if it cannot be determined from the manifest.

The workspace name can be entered in the text field labeled Workspace File or can be selected using the “...” button to the right of the text field. If the user selects a workspace name that is an existing file, the will prompt for permission to overwrite the file. A default workspace name will be chosen based on the Java file or class name and the current working directory at the time RootCause was started.

The main class name can be entered in the text field labeled Class Name. The main class must exist in the `.jar` file. If the `.jar` file contains a manifest, the main class will default to the one specified by the manifest.

The class path can be specified in the Class Path text field or it can be edited using the “...” button to the right of the text field. This will open the Class Path Dialog to edit the path. The class path value will be used to find the specified Java class if no Java file name is specified.

The JRE path can be specified in the JRE Path text field or it can be edited using the “...” button to the right of the text field. This will open the Java Path Dialog to select the path. The JRE path value will be used to find the Java Virtual Machine to run the Java program.

The Working Directory field indicates the current directory when the java command was run. This is needed in order to correctly evaluate relative path names in the Class Path.

The J2EE Server Directory can be entered or browsed to using the “...” button to the right of the text field. Enter the directory where deployable Enterprise Java Bean (EJB) and Servlet classes and jars reside. RootCause will automatically add EJB and Servlet classes and jars that are specified in any J2EE compliant XML deployment descriptors. For more information see ["RootCause J2EE Support" on page 10-14](#).

Buttons

Once valid workspace parameters have been selected, click the *OK* button to create the workspace. Use the *Cancel* button to dismiss the dialog without creating a new workspace. Use the *Help* button to figure out what is going on and how to get things done.

Reset Program Dialog

The *Reset Program Dialog* permits the user to reset the program associated with a workspace when the program is rebuilt or moved to a new location. The workspace will reconcile the changes to the program with the trace setup selected, discard any invalid trace options, and define the program which will be traced. A single program is associated with each workspace.

Program File: The program name can be entered in the text field labeled *Program File* or can be selected using the “...” button to the right of the text field. The program must exist and must have execute permission. The field will be initialized with the current program.

Main Class: If the program is a Java JAR file, the main class must be specified. The main class name can be entered in the text field labeled Main Class. The main class must exist in the JAR file. The field will be initialized with the current main class.

Buttons

Once a valid Java main class has been selected click the *OK* button to reset the program. Use the *Cancel* button to dismiss the dialog without resetting the program. Use the *Help* button to figure out what is going on and how to get things done.

Add UAL Dialog

The *Add Ual Dialog* configures a user-defined **UAL** and adds it to the workspace. Select **Add UAL** from the *Setup Menu* menu to open this dialog. Once a UAL is added to the workspace it can be enabled and disabled via the checkbox, and other options can be specified by double-clicking on it to open the *UAL Options Dialog*.

Note: This is an advanced feature, and users are encouraged to contact OC Systems for support.

Plug-In Class: If the UAL to be added requires a separate interface for configuration and operation within RootCause, that Java class name should be specified in the text field labeled *Plug-in Class*. Most UALs will not require a special interface and this field can be left blank. The *Java Exceptions Configuration Dialog* is an example of such a plug-in class.

UAL File: Specify the file (path) containing the UAL in the text field labeled *UAL File*. This identifies the actual UAL to be added to the workspace.

Name of UAL: Specify the name to use to display the UAL in the workspace tree in the text field labeled *Name of UAL*. Normally this will match the UAL file name.

UAL Description: Specify a brief description of the UAL in the text field labeled *UAL Description*. This will also be displayed in the workspace tree. This description may highlight any special configuration of the UAL.

Copy UAL: Check the box labeled *Copy UAL* to copy the UAL to the workspace. This is the recommended method of ensuring the UAL is available on a remote computer if the workspace is deployed.

Requires Trace UAL: Check the box labeled *Requires Trace UAL* if the new UAL requires that the Trace UAL be active when it is active.

Aprobe Parameters: Specify the string to follow the “-p” option after the **UAL File** on the aprobe command-line. Do not include the “-p” itself. For example, to specify a configuration file in the workspace, you might enter:

```
-c $RC_WORKSPACE_LOC/events.cfg
```

Aformat Parameters: Specify the string to follow the “-p” option after the **UAL File** on the aformat command-line. Do not include the “-p” itself. You can

refer to files in the workspace itself with the [RC_WORKSPACE_LOC](#) environment variables.

Buttons

Use the *OK* button to add the new UAL as specified. Use the *Cancel* button to abandon the operation. Use the *Help* button to figure out what is going on and how to get things done.

RootCause Options Dialog

The [Options](#) item in the [Setup Menu](#) opens the *RootCause Options Dialog*, from which most options for building, running, formatting and displaying your traces are specified. It consists of a number of tabs, but the [RootCause Options Tab](#), shown initially, shows most of the items of interest.

Buttons

Use the *OK* button to accept all changes to the RootCause options. Use the *Cancel* button to discard any changes to the RootCause options. Use the *Help* button to figure out what is going on and how to get things done.

RootCause Options Tab

The RootCause Options Tab is used to define a number of values that control the collection and display of that trace data.

Data Collection Options

The following options control data collection at the time the program is run. These are recorded per-workspace and translate into options on the *aprobe* command-line, set using the [Aprobe Options Tab](#).

Keep logged data for N previous processes: Set the maximum number of previous or concurrent processes for which data is kept by modifying the text field labeled *Keep logged data for N previous processes*. This controls for how many previous processes logged data files are retained. Larger values will use more disk space. This value must be 1 or greater, resulting in two process's data being saved: the most recent run and one previous to that.

Data File Size: Set the maximum data file size by modifying the *Data File Size* text field. This controls how large each data file can become while logging before a new data file is created. Smaller values will lead to faster times to display data,

and allow more control over the amount of data to be viewed at one time, but limit the size of tables and other monolithic data items logged by some probes.

Wraparound data logging wraps at N: Set the maximum size of wrap-around data kept in data files by modifying the text field labeled *Wraparound data logging wraps at N*. This controls how much data is kept in the wrap-around data files. Larger values will use more disk space. This number, divided by the Data File Size specified, yields the number of data files kept in the “APD ring” in the absence of snapshots. This corresponds to the *aprobe -n* argument specified in the *Number of APD Files* field in the *Aprobe Options Tab*.

Total logged data limit per process: Set the maximum size of all logged data in data files and snapshot files by modifying the text field labeled *Total logged data limit per process*. This controls the total amount of logged data that is retained in wrap-around data files plus older data files preserved as a result of a [snapshot](#) action. Larger values will use more disk space.

Data Display Options

The following options control the display of data through the [Trace Index Dialog](#) and the [Trace Display](#). These options are saved as preferences on a per-user basis.

Maximum number of items in Trace Index: Set the maximum number of items allowed in the [Trace Index Dialog](#) by modifying the text field labeled *Maximum number of items in Trace Index*. Larger values will lead to longer times to build an index.

Maximum number of events in Trace Display: Set the maximum number of events allowed in the Trace Display event tree by modifying the text field labeled *Maximum number of events in Trace Display*. Larger values will lead to longer times to display event trees.

Display N data files before selected: Set the number of data files displayed before an event selected in the [Trace Index Dialog](#) by modifying the text field labeled *Display N data files before selected*. This controls how many events are displayed before the selected event. Larger values will lead to longer times to display data.

Display N data files after selected: Set the number of data files displayed before an event selected in the [Trace Index Dialog](#) by modifying the text field

labeled *Display N data files after selected*. This controls how many events are displayed after the selected event. Larger values will lead to longer times to display data.

Build Options Tab

The *Build Options Tab* in the [RootCause Options Dialog](#) is used to set options that control how Apc files are compiled and to include additional custom Apc files. **Note: these options do not apply to Java-only workspaces.**

Aprobe Options Tab

The *Aprobe Options Tab* is used to set options that control how Aprobe collects data from the defined traces.

APD File: Choose the name of the APD file where Aprobe collects data in the *APD File* text field. Any path and name ending in “.apd” can be specified here, and the APD files will be created with that name. Note that this path is ignored for remote (deployed) workspaces. This corresponds to the aprobe **-d** option.

Number of APD Files: Set the number of APD files that will be chained together to hold trace information in the *Number of APD Files* text field. Using multiple files prevents any one file from becoming too large. As each APD file fills up it is closed and a new one is opened. If the maximum number of files have been created, the oldest one is deleted. This is an [APD ring](#). You may select 1 or more files. This corresponds to the aprobe **-n** option.

Size of APD Files: Set the maximum size in bytes of the APD files in the *Size of APD Files* text field. This value can be used to restrict the amount of information saved in each file. You can select any size from 1MB to 256MB. This corresponds to the aprobe **-s** option.

Number of APD Rings: Specify the number of APD file rings *to be preserved* in the *Number of APD Rings* text field. The number of APD rings determines how many [Process Data Sets](#) are preserved for the program *in addition to* the most recent. You need one set of rings for each simultaneous program execution you want to trace, with a minimum of 1. This corresponds to the aprobe **-k** option.

Number of Snapshot Files: Specify the number of snapshot data files in the *Number of Snapshot Files* text field. This value determines how many snapshot files are kept for the program in addition to the wraparound APD files. This corresponds to the aprobe **-t** option.

Additional Aprobe Options: Specify any additional options in the *Additional Aprobe Options* text field, just as you would on the Aprobe command line. The most commonly needed one “-qstack_size=1000000” to increase the Aprobe stack size.

Apformat Options Tab

The *Apformat Options Tab* is used to set options that control how the apformat command formats data collected from the defined traces and probes.

Additional Apformat Options: Specify any additional options in the *Additional Apformat Options* text field, just as you would on the Apformat command line.

Run Options Tab

The *Run Options Tab* in the [RootCause Options Dialog](#) is used to set options required to run the program on the local computer using the [Run Program](#) menu item or the [Run](#) button. These options are ignored unless your application is run directly from the RootCause GUI.

Working Directory: Set the path of the working directory where the program should run in the *Working Directory* text field. This also specifies the directory from which the load modules are evaluated.

Command To Invoke Program: This is the main class name that will appear on the Java command; *do not change it*. If you need to run the program with something other than the “java” command you cannot use the Run button or menu item.

Program Parameters: Specify any program parameters in the *Program Parameters* text field, just as you would on the command line.

Java Parameters: Specify any parameters to the Java Virtual Machine (JVM) in the *Java Parameters* text field as you would on the command line.

Source Options Tab

The *Source Options Tab* is used to set options that control how missing source files are handled in the Trace Setup and Trace Display dialogs.

Don't Prompt for Source Files: Check the *Don't Prompt for Source Files* box to specify that actions in the [Trace Setup Dialog](#) and [Trace Display](#) window are to ignore missing source files. This setting is also available in the source file

prompt dialog itself. You may find source files later using the *Find Source File* menu item.

Edit Source Path Dialog

The *Edit Source Path Dialog* is opened by the *Source Path* item in the *Setup Menu*, and allows the user to edit the path used to search for source files. Source files are displayed in the *Trace Setup Dialog* and *Trace Display* to provide context information when selecting traces and viewing trace events.

The list displays, in order, the directories searched for source files if the full path recorded in the object or class file is not found.

Buttons

Add: Use the *Add* button to add a new path before the selected one. A file chooser will open to select the path.

Move Up: Use the *Move Up* button to move the selected path ahead of the path above it.

Move Down: Use the *Move Down* button to move the selected path behind the path below it

Remove: Use the *Remove* button to remove the currently selected path.

Use the *OK* button to accept the changes to the source path. Use the *Cancel* button to discard any changes to the source path. Use the *Help* button to figure out what is going on and how to get things done.

Edit Class Path Dialog

The *Edit Class Path Dialog* is opened by the *Class Path* item in the *Setup Menu*, and allows the user to edit the path searched for classes found in the *Trace Display* events, and also when running Java using the *Run* button.

The list displays, in order, the directories and/or JAR files that are searched by RootCause or the **JVM** itself.

Buttons

Add: Use the *Add* button to add a new path before the selected one. A file chooser will open to select the path.

Move Up: Use the *Move Up* button to move the selected path ahead of the path above it.

Move Down: Use the *Move Down* button to move the selected path behind the path below it

Remove: Use the *Remove* button to remove the currently selected path.

Use the *OK* button to accept the changes to the class path. Use the *Cancel* button to discard any changes to the class path. Use the *Help* button to figure out what is going on and how to get things done.

Java Path Dialog

The *Java Path Dialog* is opened by the [JRE Path](#) item in the [Setup Menu](#), and edits the path to the Java Runtime Environment ([JRE](#)) used to execute Java programs.

Type the full path to the JRE root directory in the text field labeled JRE Path, or use the “...” button to open a file chooser to select the path.

Buttons

Use the *OK* button to accept the changes to the JRE path. Use the *Cancel* button to discard any changes to the JRE path. Use the *Help* button to figure out what is going on and how to get things done.

UAL Options Dialog

The *UAL Options Dialog* is opened by the [Edit UAL](#) item in the [Setup Menu](#) or [Workspace Tree Workspace Tree Popup Menu](#), or by double-clicking on a name in the [UALs node](#) node in the Workspace Tree.

This is the “default UAL plug-in” for UALs added using [Add UAL](#). It lets you set options for [UALs](#) associated with the workspace. It is overridden by more powerful configuration dialogs for the trace and [java_exceptions](#) UALs. Contact OC Systems for more information on writing interfaces for UALs.

The UAL file and name are displayed in the text fields labeled *UAL File* and *UAL Name*. These values cannot be changed.

If a UAL requires command-line parameters at *Aprobe-time* or *Apformat-time*, you can change those values in the text fields labeled *Aprobe Parameters* and

Apformat Parameters. The “-p” option used on the *aprobe* and *apformat* command-line to introduce UAL arguments should not be specified here.

Buttons

Use the *OK* button to set the Ual options as specified. Use the *Cancel* button to abandon the operation. Use the *Help* button to figure out what is going on and how to get things done.

Java Exceptions Configuration Dialog

The `java_exceptions` predefined UAL in the *Workspace Tree* includes a “configuration plug-in” interface which allows the user to change the default actions associated with Java exceptions when the Java Exceptions UAL is enabled. This dialog is opened by double-clicking on the `java_exceptions` label, or selecting *Edit UAL* from the *Workspace Tree Popup Menu* or the *Setup Menu*.

There are two levels of exception reporting provided, “Logging” and “Snapshots”.

When an exception is Logged, an Exception event and traceback is logged which appears in the *Trace Index Dialog* if Exception events are selected in the *Select Events Dialog*, and an Exception marker appears in the *Trace Display*.

When an exception Snapshot is taken, in addition to the simple event logged, a `snapshot` event is created, which also appears in the *Trace Index Dialog*.

By default, all user-defined exceptions are Logged. In addition, common Java Runtime and RMI exceptions have Snapshots taken by default.

The *Java Exceptions Configuration Dialog* allows changing these defaults. There are two main panes in the dialog, one for Logging and one for Snapshots.

Logging Exceptions: Within the *Logging Exceptions* pane, exceptions can be excluded from logging by adding the full Java exception class name on a separate line of the multi-line text field.

Exception Snapshots: Within the *Exception Snapshots* pane, there are three sub-panes:

Default: The first, labeled *Default*, allows turning on or off of snapshots for Runtime and RMI exceptions. The default is “on” (checked) for both types.

Include: The second sub-pane under Exception Snapshots is labeled *Include*, which allows the addition of individual exception classes for which a snapshot is to be taken. Specify the full name of each exception to be included on a separate line.

Exclude: Finally, the *Exclude* sub-pane identifies individual exception classes for which a snapshot is *not* to be taken. Specify the full name of each exception to be excluded on a separate line.

Buttons

Use the *OK* button to update the configuration to be applied to the next run of the program. Use the *Cancel* button to abandon any changes to the workspace options.

Note: No configuration dialog is available for [exceptions](#) due to the much more limited use of exceptions in C++. However this plug-in mechanism is designed to allow site-specific configuration and extension. Contact OC Systems for more information about such customizations.

Run Program Dialog

The *Run Program Dialog* is shown by the [Run](#) button (the [Run Program](#) operation) to run a traced program on the local computer. This is provided as a convenience for running simple programs. Programs that require special start-up scripts or batch files can be run from the command line in a normal fashion (as long as they have been registered with a RootCause workspace for tracing). The working directory, program path, and program parameters (as specified in the Program Options Dialog) are displayed.

Choose the *Autoload Output* checkbox to automatically open a [Trace Data Dialog](#) upon program termination to format and display the trace data.

Buttons

Use the *OK* button to run the program as specified. Use the *Cancel* button to abandon running the program. Use the *Help* button to figure out what is going on and how to get things done.

Deploy Dialog

The *Deploy Dialog* is opened by the [Deploy](#) button (the [Deploy Program](#) operation) to deploy the traces for a program to a remote computer so the program can

be traced there. The dialog allows the user to select some final options, and to review other options.

Output file: Type the path name of the output file to be created in the text field labeled *Output file* or select a path using the “...” button. This is the file that must be transmitted to the remote computer for installation.

License file: Type the path name of the license file to be used in the text field labeled *License file* or select a path using the “...” button. This is normally `$APROBE/licenses/agent_license.dat`.

If the License file does not have the proper name or cannot be found an error is given and the user must provide a satisfactory file. If the file does not contain a valid license, an error is given but the user may continue and create a deploy package, but then a license must be provided at the remote site.

Aprobe Options: Use the *Aprobe Options* tab to inspect the Aprobe options that will be used. If they are not correct, go back to the [RootCause Options Dialog](#) to change them.

Program Options: Use the *Program Options* tab to verify the default path to the program with which the workspace is associated. The name of the program file is not referenced in the remote environment, but the name of the main class within it must be the same

UALs: Use the *UALs* tab to inspect which UALs will be activated when the program is traced. If they are not correct, go back to the [UALs node](#) node in the [Workspace Tree](#) of the main window to change them.

ADI Files: Use the *ADI Files* tab to select for which modules you want to generate Aprobe debug information (ADI) files. Only compiled modules are listed here -- none are needed for Java.

Buttons

Use the *OK* button to create the deploy file from the given parameters. Use the *Cancel* button to abandon the deploy operation. Use the *Help* button to figure out what is going on and how to get things done.

Decollect Data Dialog

The *Decollect Data Dialog* is opened by the *Decollect* button (the *Decollect Collected* operation) to unpack data collected on a remote computer for examination on the local computer.

When the operation completes, an *Open Decollection* is automatically done on the newly created directory to allow the data to be formatted and viewed.

Collect File: Type the path name of the collect file to unpack in the *Collect File* text field or use the “...” button to open a file chooser to select the file. This is the file that was created by a collect operation on the remote computer and transmitted to the local computer.

Destination To Create: Type the path name of the destination directory to be created in the *Destination To Create* text field or select a path using the “...” button. This is the directory that becomes the *decollection* that will contain the unpacked data on the local computer.

Buttons

Use the *OK* button to unpack the collect file to the chosen destination. Use the *Cancel* button to abandon the decollect operation. Use the *Help* button to figure out what is going on and how to get things done.

Trace Setup Dialog

The *Trace Setup Dialog* is used to set up traces and probes for the program. It consists of three parts: the *Program Contents Tree* on the left, the *Source Pane* on the upper right, and the *Variables Pane* and *Probes Pane* as tabs on the lower right. The following sections describe each of these in more detail.

Program Contents Tree

The *Program Contents Tree* displays the currently selected traces for the program. At the root is the program node. Under this node are the libraries (*modules*) against which the program has been linked. Additional *dynamic modules* that have been explicitly added to the workspace using *Add Dynamic Module* in the main *Workspace Menu* will also appear.

Nodes with child nodes beneath them have a “lever” next to that node. Clicking on a “lever” next to a node in the tree, or double-clicking on node’s label, will expand it, showing the immediate children of that node. Doing these actions on an already-expanded node will collapse it, hiding its child nodes. Double-click also expands or collapses a node.

Java Program Contents

Java program contents are organized as follows:

P Main Class - the program node based on main class name

M Root Java Module (\$java\$) - root of all Java methods

J Class Path Element - JAR or directory in classpath

Package Element - e.g., com or sun, if applicable

C Class - a class in the parent JAR/directory/package

m Method in Class

Class Path elements are listed in alphabetical order, not the order they appear in the classpath. Use the [Edit Class Path Dialog](#) to view or change the run-time order. The Class Path element is *not* significant in a [Trace All In](#) operation, only the package and class names. So selecting the “com” node under pet-store.jar and clicking Trace All In com will really trace all classes in all packages that start with com in the *whole application*, not just those in pet-store.jar.

Static constructor methods have names <clinit>() and user-defined constructors are <init>(args).

Black dots next to the methods indicate that the method will be traced. *Blue dots* next to the methods indicate that [actions](#) such as logging are defined.

Selecting a method node will cause the source to be displayed in the source pane, if possible. In the [Variables Pane](#), a single checkbox for logging all the parameters will be displayed. The probe triggers and actions associated with the method will be displayed in the [Probes Pane](#).

Trace Setup Popup Menu

Operations on nodes in the [Program Contents Tree](#) are done via a popup menu. Select a node in the tree and use the right mouse button (MB3) to display the popup menu.

Trace This Item: Use *Trace This Item* to add a trace for the selected method node.

Don't Trace This Item: Use *Don't Trace This Item* to remove a trace (black dot) for the selected method node.

Enable Load Shedding for This Item: Use *Enable Load Shedding for This Item* to re-enable [load shedding](#) (the default action) for the selected method node. This action is available only when load shedding has been previously disabled, as indicated by a red dot.

Disable Load Shedding for This Item: Use *Disable Load Shedding for This Item* to disable [load shedding](#) for the selected method node. This is generally not necessary unless the function has a very high execution rate, yet you still want to trace it, at the risk of slowing the overall trace. Disabling load shedding causes the item to be marked with a red dot.

Trace All In: Use *Trace All In* to add a trace for all the child nodes of the selected module, file, directory or class node.

Note: Any enclosing JAR or directory is *not* significant in a *Trace All In* operation, only the package and class names. So selecting the “com” node under `pet-store.jar` and clicking *Trace All In* com will really trace all classes in all packages that start with com in the *whole application*, not just those in `pet-store.jar`.

Don't Trace All In: Use *Don't Trace All In* to remove traces (black dots) for all the child nodes of the selected module or class node.

Remove Probes For All Child Items: Use *Remove Probes For All Child Items* to remove any probes applied to the selected node and all its child nodes. Note that probes here are those actions indicated by the blue dots, which were added from the [Variables Pane](#) or [Probes Pane](#).

Trace All Lines in Function: Use *Trace All Lines In Function* to add a trace for all the lines in the selected method node.

Don't Trace All Lines in Function: Use *Don't Trace All Lines In Function* to remove traces for all the lines of the selected method.

Edit Wildcards: Use *Edit Wildcards* to examine and change the TRACE and REMOVE directives for the module containing the selected item, using the [Edit Wildcard Strings Dialog](#).

Find Function/Method: Use *Find Function/Method* to find a function or method in the program contents tree. This will open the [Find In Program Contents Dialog](#). This is the same as the [Find](#) button at the bottom of the window.

Find Source File: Use *Find Source File* to locate the source file for the selected class or method.

Source Pane

The *Source Pane* displays the source file for the currently selected method in the *Program Contents Tree*. The source for the current method is annotated with line numbers and checkboxes which indicate which lines can be traced in the method. Checking a source line will cause the applicable probe triggers and actions to be displayed in the *Probes Pane*.

Variables Pane

The *Variables Pane* displays the parameters that can be logged:

- on entry to a method; or
- on exit from a method.

When a method is selected in the *Program Contents Tree*, the *Variables* pane will show that parameters can be logged on entry to a method, which also implies logging the return value on exit.

Probes Pane

The *Probes Pane* displays the probes that can trigger [actions](#):

- on entry to a method; or
- on exit from a method.

When a method is selected in the *Program Contents Tree*, the *Probes* pane will show probes activated on entry to or exit from the method.

When the program node in the *Program Contents Tree* (denoted by the letter **P**) is selected, the *Probes Pane* displays probes that can trigger actions:

- on program entry;
- on program exit;
- on thread entry; or
- on thread exit.

To define a probe, first check the “On” box on the left to activate it. Then select a trigger from the *Probe Trigger* options menu.

Next select an action from the Probe Action options menu. If the action requires a parameter, specify it in the Probe Parameter text field or combo box. You can disable a probe by unchecking the On check box associated with the probe.

Probe Actions

The following actions may be selected from the Probe Actions options menu:

Log Comment: Log a string literal at the given Trigger point. The Parameter to this action is the string to be printed. It will appear as a COMMENT event in the [Event Trace Tree](#).

Log Traceback: Log a stack traceback at the given Trigger point. The Parameter is the maximum depth to trace back. The overhead of the traceback operation is proportional to this depth. It will appear as a TRACEBACK node in the Event Trace Tree.

Log Statistic: Log time or other information specified by the Parameter. The statistics appear as a *Process Statistics* node in the Event Trace Tree. The Parameter values are:

- `gethrtime(3C)` (wall time) - calls the `gethrtime()` system function and displays the value returned, the elapsed time since the start of the program.
- `gethrvtime(3C)` (CPU time) - calls the `gethrvtime()` system function and displays the value returned, the CPU time consumed by the process.
- `rusage(3C)` (resource usage) - calls the `rusage()` system function, and displays the fields of the structure it returns.

Enable Tracing: Enable tracing that was disabled by an earlier Disable Tracing action. Has no effect if tracing was already enabled.

Disable Tracing: Disable tracing at the trigger point, to reduce the amount of data logged. It is useful to use Disable Tracing at the On Entry trigger point of a method, and Enable Tracing at the On Exit point, or vice versa, to control the data logged.

Log Snapshot: Cause a data [snapshot](#) to occur at the probe point, which also logs a SNAPSHOT event. These events are shown in the [Trace Index Dialog](#), making them very easy to locate even if a lot of data has been logged. The

“Probe Parameter” field is a text field which defaults to `ROOTCAUSE_SNAPSHOT`, but which may be replaced with any reasonably short text string to more uniquely identify the snapshot point.

Buttons

Find: Use the *Find* button to search the program contents tree for methods of interest. This will open the *Find In Program Contents Dialog*.

Options: Use the *Options* button to select advanced options that affect all traces. This opens the *Global Trace Options Dialog*.

Custom: Use the Custom button to get a template of the probe deployment descriptor corresponding to your custom Java file. This opens the *Generate Custom XMJ Dialog*. There is no further automated support for including custom Java. See [Chapter 11, "Custom Java Probes"](#).

OK: Use the *OK* button to build a UAL from the selected trace setup, and then dismiss the *Trace Setup Dialog*.

Apply: Use the *Apply* button to build a UAL from the selected trace setup. The *Trace Setup Dialog* will stay visible.

Dismiss: Use the *Dismiss* button to close the *Trace Setup Dialog* without building a UAL from the trace setup.

Help: Use the *Help* button to figure out what is going on and how to get things done.

Find In Program Contents Dialog

The *Find In Program Contents Dialog* is used to search for methods containing a specific pattern, or to find a method based on its source file name and line number.

Find String: When the *Find String* tab is selected, you provide a string that matches all or part of a method in the *Program Contents Tree*. You must select a module, class, or method node in the program contents tree to indicate the start of the search. The next method that contains the given search string will be selected, or a dialog will indicate that no matches were found.

Consider Case: If *Consider Case* is selected, the search for the given string will exactly match the case of letters in the given search string. By default lower and upper case of the same letter are considered equal.

Search All Modules: If *Search All Modules* is selected, then all modules will be searched without asking for confirmation after each one is searched. You can select alternate starting nodes while the search dialog is visible to direct the search.

Goto File: The *Goto File* tab is used to search based on a source file name, and optionally a line number, starting from the first method in the module. You provide the full or simple name of a file to search for. The name provided is compared to the end of the full pathname of the source file containing each method, so you must always provide the file extension in your search string.

Line Number: If no *Line Number* is specified, the first method associated with the given file is identified. If a *Line Number* is specified, the method containing that line in that file is found, if any, or else the method in the file whose start line is closest to the given line.

Buttons

Use the *Next* button to find the next occurrence of the string in a function or method node. Use the *Previous* button to find the previous occurrence of the string in a function or method node. Use the *Goto* button to find the specified line number in a file. Use the *Cancel* button to dismiss the search dialog. Use the *Help* button to figure out what is going on and how to get things done.

Global Trace Options Dialog

The *Global Trace Options Dialog* is used to set advanced options to control the trace and logging of data. It is opened by clicking the *Options* button in the *Trace Setup Dialog*

Dereference Pointers: The *Dereference Pointers* option determines whether data whose type is a pointer type has the value of the pointer (the address), or the referenced data logged. Check the option to log the referenced data.

Log Java Class Loads: The *Log Java Class Loads* option determines whether each load of a Java class in the application is logged. This is enabled by default, and adds little overhead, but you may want to disable it if these events aren't helpful.

Maximum Logged String Length: The *Maximum Logged String Length* text field determines how many bytes of dereferenced string types are logged. Lowering this value can reduce the amount of data logged and reduce the impact that tracing has on the program's performance.

Enable Load Shedding: This checkbox indicates whether [load shedding](#) will be enabled on the next trace. The scale and text field underneath this checkbox indicates the relative amount of tracing overhead that should be tolerated before tracing is disabled on a method. This is recorded on a per-workspace basis, and is enabled by default to allow moderate overhead.

Individual methods may be excluded from load shedding using the [LOAD_SHED Table](#) associated with a LOAD_SHED node at the end of the [Trace Display](#).

Buttons

Use the *OK* button to accept the option values as displayed. Use the *Cancel* button to discard any changes and leave the values as they were. Use the *Help* button to figure out what is going on and how to get things done.

Edit Wildcard Strings Dialog

The *Edit Wildcards* item on the [Trace Setup Popup Menu](#) in the [Program Contents Tree](#) brings up the *Edit Wildcard Strings Dialog*.

The contents of this dialog reflect the Traces selected from within the Trace Setup popup menu for the currently selected module. The module to which the dialog contents apply is shown in the dialog title.

Trace Wildcards: The *Trace Wildcards* list on the left specifies those methods that will be traced.

Don't Trace Wildcards: The *Don't Trace Wildcards* list on the right shows those methods that will be explicitly removed from the list of methods to be traced, so the final set of traces is the difference between the two lists.

Add: To add a wildcard to a list, enter it in the text field below that list, then click the *Add* button.

Update: To replace an existing item with the contents of the text field, select that item and click the *Update* button.

Remove: To remove an item from the list, select that item and click the *Remove* button.

More Buttons

Click the *OK* button to save the lists as shown. Click the *Cancel* button to discard any changes. Click the *Help* button to view this text.

The names in the list are “probe names” of the form

```
"class::method(String)"
```

There is only one wildcard character, '*', and '*' may appear only as the first and/or last character in the wildcard string. The following are examples of valid wildcards:

```
"*"          all methods in the module
```

```
M*          all methods whose name starts with M
```

```
ErrorClass::*  
            all methods in class ErrorClass
```

Generate Custom XMJ Dialog

The Generate Custom XMJ Dialog is opened when the [Custom](#) button at the bottom of the Trace Setup dialog is clicked. It presents a text dialog describing how to construct a custom probe, including the exact XMJ text applicable to the selected method. See [Chapter 11, "Custom Java Probes"](#) for more information.

New Class Dialog

The *New Class Dialog* is opened when an attempt to find a class in the Trace Setup dialog fails. Use this dialog to enter the path for class file to be added to the Trace Setup. You can browse for .class and .jar files that match the class name. If the class name field is already filled in, you will not be able to modify it. If a matching class file was found in the classpath, it will be automatically displayed but you may browse to a different one if necessary.

Buttons

Use the *OK* button to select the class file. Use the *Cancel* button to close the dialog without selecting a class file. Use the *Help* button to figure out what is going on and how to get things done.

Trace Data Dialog

The *Trace Data Dialog* permits the user to select the exact data files to be [formatted](#) for display in the *Trace Display*. The user can make this choice based on the data present in each data file, to reduce the total size of the display. Data from additional [workspaces](#) can be made available for selection.

Data files available for selection are displayed in the tree labeled *Trace Data Files*. Check the checkbox to the left of a node in the tree to select that data file for formatting and display. Nodes in the tree represent data files collected for processes running traced applications, the [RootCause Log](#) file, or a [decollected log](#) file.

Buttons

Add Process: Use the *Add Process...* button to add the data files collected from another process running a traced application to the data files tree. This will open the [Add Process Data Dialog](#) to select data files. Once added, the data files can be selected for format and display.

OK: Once the data files have been chosen, click the OK button to format and display the data in a new *Trace Display*.

Apply: Click the *Apply* button to format and display the data in the original Trace Display. This is only available when the dialog is opened from a Trace Display.

Cancel: Use the Cancel button to dismiss the dialog.

Help: Use the Help button to figure out what is going on and how to get things done.

Add Process Data Dialog

The *Add Process Data Dialog* permits the user to select [Process Data Set](#) from either the *Trace Data Dialog* or *Select Data Files Dialog*. A Process Data Set is a directory containing all data files collected in a single run of an application. The user can navigate through the file system to select the relevant data sets. Multiple Data Sets may be available from a single directory because the file chooser searches the subdirectories of the selected directory recursively.

Look In: The directory to search can be entered in the text field labeled *Look In* or can be selected using the “...” button to the right of the text field.

Process Data Sets: The available data sets will be displayed in the list labeled *Process Data Sets*. Select a data set by clicking on it.

Process Data Set: The name of the selected data set will be displayed in the text field labeled *Process Data Set*.

Buttons

Once a Process Data Set has been chosen, click the *OK* button to add the data to the parent dialog. Use the *Cancel* button to dismiss the chooser. Use the *Help* button to figure out what is going on and how to get things done.

Trace Index Dialog

The *Trace Index Dialog* permits the user to generate an index for selected data files.

The index contains special marker [events](#) selected by the user. The index can allow the user to quickly find notable events in large traces and to examine all the events surrounding the marker in the [Trace Display](#).

There is always one event in the index, corresponding to the Last Data Recorded, and which always appears first since the initial order of events is most-recent first.

Double-click on an event to immediately open a new [Trace Display](#) containing just that event and its context.

Click on an event in the table to select it for display. Use Shift-click to select a contiguous block of events. Use Ctrl-click to select multiple, non-contiguous events.

What is Indexed

The Trace Index Dialog is opened as a result of a number of different actions, which generate an index of different data:

1. Indexing Current Workspace Data

When you:

- choose [Index Process Data](#) in the [Analyze Menu](#); or
- click the [Index](#) button in the Workspace buttonbar;

you have chosen to display an index for the *current data* in the work-

space. This will *replace* any index you have constructed already unless the data is unchanged from the last time these operations were performed.

2. Indexing Decollected Data

When you choose [Open Decollection](#) or [Recent Decollections](#) in the [Workspace Menu](#), an index is built using the most recent data in the decollection.

3. Re-Indexing Displayed Data

When you choose [Add From Index To Display](#) in the Trace Display [File Menu](#), the index associated with the data currently being displayed is shown. This means that you can save and modify previously constructed indexes by leaving a corresponding Trace Display window open.

4. Indexing Selected Process Data

When you:

- choose [Add Selected Process Data](#) in the Trace Display [File Menu](#) or the [Trace Display Popup Menu](#) (when an APP_TRACED event is selected); or
- double-click on a [Data node](#) in the Workspace Tree;

the data associated with the selected process is indexed.

In all cases, you can use the [Select Data Files](#) button to change the [Data Files](#) that are indexed.

Index Columns

Click on a column header in the table to sort the table by that column. Click again to reverse the order of the sort.

The following columns are shown in the [Trace Index Dialog](#):

Time: the [timestamp](#) of the event. The newest event is shown first initially.

Application: the name of the file associated with the [application](#). This is the same as the Program name associated with the [workspace](#) that caused the data to be recorded.

Process: the process id ([PID](#)) of the process in which the event occurred.

Thread: the Thread ID of the thread in which the event occurred. These thread IDs are internal to Aprobe, and may not correspond to those shown by other debugging tools.

Kind: the kind of [event](#), SNAP or EXCP, corresponding to the [Snapshots](#) and [Exceptions](#) checkboxes in the [Select Events Dialog](#). For each [Process Data Set](#) there is also one FILE kind event with the label Last Data Recorded, which is the newest event from that process.

Event: the label associated with the [event](#). This corresponds to the Probe Parameter associated with a [Log Snapshot](#) probe, or maybe be a special value used by a predefined probe, such as (Java) EXCEPTION, C++ EXCEPTION, or Ada EXCEPTION.

Details: the first text line of details associated with the event.

Buttons

Refresh Index: Use the *Refresh Index* button to regenerate the index using the current selections.

Select Data Files: Use the *Select Data Files...* button to choose which data files are scanned to generate the index. This will open the Select Data Files Dialog.

Select Events: Use the *Select Events...* button to choose kind of events to include in the index. This will open the Select Events Dialog.

Find In Index: Use the *Find In Index...* button to find an event or events in the index that match(es) a string. This will open the Find Events Dialog.

OK: Once the events have been chosen, click the OK button to format and display the data in a new Trace Display. Alternatively, you can double-click an event to display it.

Apply: Click the *Apply* button to format and display the data in the original Trace Display. This is only available when the dialog is opened from a Trace Display.

Cancel: Use the *Cancel* button to dismiss the dialog.

Help: Use the *Help* button to figure out what is going on and how to get things done.

Select Data Files Dialog

The *Select Data Files Dialog* permits the user to select the exact data files to be scanned when generating the index in the *Trace Index Dialog*.

Data files available for selection are displayed in a tree. Check the checkbox to the left of a node in the tree to select that data file for indexing. Nodes in the tree represent data files collected for processes running traced applications, the Root-Cause log file, or a decollected log file.

Buttons

Add Process Data: Use the *Add Process Data* button to add the data files collected from another process running a traced application to the data files tree. This will open the *Add Process Data Dialog* to select data files. Once added, the data files can be selected for indexing.

Update: Once the data files have been chosen, click the *Update* button to generate the index and close the dialog.

Change: Click the *Change* button to accept the changes to the data files, close the dialog, but not generate the new index.

Cancel: Use the *Cancel* button to dismiss the dialog.

Help: Use the *Help* button to figure out what is going on and how to get things done.

Select Events Dialog

The *Select Events Dialog* permits the user to select the kind of events included in the *Trace Index Dialog* index.

Check the checkbox to the left of an event kind to select those events for inclusion in the index:

Snapshots: include SNAP events in the index, recorded by the `java_exceptions` UAL or by a user-inserted *Log Snapshot* probe.

Exceptions: include EXCP events in the index, recorded by the `java_exceptions` and `exceptions` UALs.

Buttons

Update: Once the event kinds have been chosen, click the *Update* button to generate the index and close the dialog.

Change: Click the *Change* button to accept the changes to the event kinds, close the dialog, but don't generate the new index.

Cancel: Use the *Cancel* button to dismiss the dialog.

Help: Use the *Help* button to figure out what is going on and how to get things done.

Find Text In Events Dialog

The *Find Text In Events Dialog* permits the user to search for events in the index which match a text string. It is launched by the [Find In Index](#) button of the [Trace Index Dialog](#).

Search For: Enter the text string to match in *Search For* text field.

Consider Case: Check the *Consider Case* checkbox to make the search case-sensitive.

Buttons

Find: Once the search string is set, click the *Find* button to find the first event that matches the string. The string starts from the event following the first selected event, or from the first event in the index if no events are selected.

Find All: Click the *Find All* button to find every event in the index that matches the string.

Cancel: Use the *Cancel* button to stop the search and dismiss the dialog.

Help: Use the *Help* button to figure out what is going on and how to get things done.

Trace Display

The *Trace Display* window is used to examine detailed trace output logged for traced programs. The Trace Display is opened from the [Trace Data Dialog](#) and from the [Trace Index Dialog](#) to show data collected for specific processes. It also displays the start of each process in the [RootCause Log](#).

The Trace Display is composed of the following parts:

- the menu bar and the [Stepping Toolbar](#) across the top;
- the [Event Trace Tree](#) on the left side;
- the [Source Pane](#) on the upper right;
- the [Event Details Pane](#), the [Call Stack Pane](#), the [Program Information Pane](#), and the [Data Files Pane](#) on the lower right.

These are described below.

File Menu

The Trace Display initially displays the contents of a single APD or log file.

Refresh: Choose *Refresh* to reformat and reload the data files or RootCause log from which the current Trace Display was built. This is useful to add the most recent events in the RootCause Log or from a data set that is still being written to.

Open Associated Workspace: Use *Open Associated Workspace* to open the workspace associated with a program in the RootCause log file. This menu item is only enabled if an APP_START or APP_TRACED event is selected in a RootCause log file.

Add Data Files To Display: Use *Add Data Files To Display* to open a [Trace Data Dialog](#) from which a different group of data files may be selected for display. This is most useful for viewing additional events before or after those currently selected. You can determine which files are currently selected by looking in the [Data Files Pane](#) at the lower right. From that dialog, you can click [Apply](#) to replace the current Trace Data events with the new ones, or OK to create a Trace Display window.

Add From Index To Display: Use *Add From Index To Display* to open a [Trace Index Dialog](#) from which different indexed events may be selected for display. From that dialog, you can click [Apply](#) to replace the current Trace Data events with the new ones, or OK to create a Trace Display window.

Add Selected Process Data: Use *Add Selected Process Data* to build an index for the data file(s) associated with the selected process. This menu item is only enabled if an APP_TRACED event is selected in a RootCause log file and data exists for the process.

Save As XML: Use *Save As XML* to save the current event tree as an XML file. The XML file can be processed outside of RootCause, or reloaded for later display.

Save As Text: Use *Save As Text* to save the current event tree as a text file. The text file can be processed outside of RootCause.

Close: When you are finished viewing the trace display use *Close* to close the Trace Display window.

Edit Menu

The Edit menu provides operations on nodes in the [Event Trace Tree](#), and are also available in a [Trace Display Popup Menu](#) there.

Deselect Function In Trace Setup: While developing a trace on a local computer you may want to remove unnecessary method traces from your trace. Select a method call in the trace event tree, then use the *Deselect Function In Trace Setup* to remove the selected function from the trace setup.

NOTE: When used on a LINE event, this will deselect the tracing of the entire function, not just that line.

Find Function In Trace Setup: On the other hand, you may decide that there is additional data that must be collected with a particular event. Select an ENTER, EXIT, LINE, or CALL_FROM method in the trace event tree, then use the *Find Function In Trace Setup* menu item to open the [Trace Setup Dialog](#) with the selected method highlighted. You can then add or remove probes of logged data items. If the class containing the method cannot be found, the [New Class Dialog](#) is presented to allow the user to locate the file in which the class is defined.

Find Class In Trace Setup: To move to the node in the Trace Setup tree corresponding to the Java class containing the method in a CALL node, select the method in the trace event tree, then use the *Find Class In Trace Setup* menu item to open the Trace Setup Dialog with the selected method highlighted. You can then add or remove probes of logged data items. If the class cannot be found, the [New Class Dialog](#) is presented to allow the user to locate the file in which the class is defined.

Find Source: Use the *Find Source* menu item to locate the source file corresponding to an ENTER, EXIT, or LINE event.

Find Function In Trace Events: You can use the *Find Function In Trace Events* menu item to start a search for the next or previous occurrence of the currently selected event in the Trace Display. Select the event node in the trace event tree, then use the menu item to open the *Find Text in Trace Events Dialog*. The search is started at the selected event.

Show Associated Table: Some event nodes, like CALL_COUNTS, have tables of information associated with them. Select the event node and use the *Show Associated Table* menu item to display the data in a *Table Dialog*.

Find Text in Trace Events: You can use the *Find Text in Trace Events* menu item to search the trace event tree for events containing a given string. The details of the trace events are searched as well. The search is started at the selected event or at the first event if one is not selected.

View Menu

The trace events are initially grouped by threads, with each showing a well-formed call tree. Use the *View Menu* to change this.

By Threads: Use the *By Threads* menu item to separate the displayed events by individual threads. This will permit you to examine the flow within each thread in isolation from the others.

By Time: Use the *By Time* menu item to display the trace events in time order. Events from different threads and processes will be interleaved to indicate the order in which they occurred. This can help you understand the interactions between the different threads.

Step Menu

Use the *Step Menu* to step through the individual events in the event trace. Stepping allows you to review program execution forwards and backwards.

Step Next Forward: Use the *Step Next Forward* menu item to step to the next event in the forward direction.

Step Next Backward: Use the *Step Next Backward* menu item to step to the previous event.

Step Into Forward: Use the *Step Into Forward* menu item to step into the next method call in a forward (increasing time) direction. Using this menu item from the beginning of a trace will visit every trace event in the order it occurred.

Step Over Forward: Use the *Step Over Forward* menu item to step over the current method call in a forward direction. This can be used to skip from an ENTER node to the corresponding EXIT node, if it exists in the current trace.

Step Out Forward: Use the *Step Out Forward* menu item to step out of the current call to the next event in a forward direction. This can be used to get out of calls that are no longer of interest.

Step Into Backwards: Use the *Step Into Backwards* menu item to step into the most deeply nested previous call. This is effectively the most immediately previous event.

Step Over Backwards: Use the *Step Over Backwards* menu item to move backwards over a call that is not of interest. This is useful to go from an EXIT node to the corresponding ENTRY node, if it exists in the current trace.

Step Out Backwards: Use the *Step Out Backwards* menu item to move backwards out of the current call to the previously traced caller.

Help Menu

Use the *Help* menu to figure out what is going on and how to get things done.

Stepping Toolbar

The *Stepping Toolbar* contains a number of buttons that help you step through the trace events in sequential forward and reverse orders. These correspond to the items in the [Step Menu](#).

The two large up and down arrows correspond to:

- *Step Next Forward*
- *Step Next Backward*.

The next bank of six buttons are:

- *Step Into Forward*
- *Step Over Forward*
- *Step Out Forward*
- *Step Into Backwards*
- *Step Over Backwards*
- *Step Out Backwards*

Find

The *Find* button provides quick access to the *Find Text in Trace Events* operation, also available from the *Edit Menu* and the *Trace Display Popup Menu*.

Event Trace Tree

The *Event Trace Tree* displays the trace events logged by the traced program. The tree can be displayed in two different ways, *By Threads* and *By Time*. The initial view presented is *By Threads*.

In the by-thread display, the tree will display one branch for each thread that was traced. Entry and exit to called methods and other events can be viewed by expanding the branches of the tree. You can use the stepping buttons to navigate through the trace events, or you can directly manipulate the trace event tree with the mouse.

In a by-time display, the tree will display slices of execution from the traced threads. Each thread slice can be expanded to display the events that occurred with that slice.

Most nodes in the tree consist of an icon, usually just a letter; followed by a label; followed by a description. Selecting most nodes also updates the contents of the *Source Pane* and the *Event Details Pane* with additional information.

The following kinds of events are displayed:

APP_START	a program was started but not traced.
APP_TRACED	a program was started and traced.
START_DISPLAYED_DATA	The start of the displayed data.
END_DISPLAYED_DATA	The end of the displayed data.
THREAD_START	the start of thread in a program.
THREAD_END	the end of a thread in a program.
PROCESS/THREAD	a thread slice.
ENTER	entry into a function or method.
EXIT	exit from a function or method.
ENTER (cont.)	the continuation of a method call trace.
CALL_FROM	the caller of the function in the immediately following ENTER node.

LINE	a source line trace.
LINE (Call)	a source line marking the caller of a method.
COMMENT	a probe logged a comment.
PROGRAM_COMMENT	a comment logged at program start.
TRACEBACK	a probe logged a traceback.
TEXT	unformatted program or probe output.
EXCEPTION	An exception-triggered snapshot
SYN_CALL_COUNTS	synthesized (event-based) call count table.
JAVA_LOAD_SHED	information about methods disabled by load shedding .
SYN_JAVA_CALL_COUNTS	synthesized (event-based) call count table
JAVA_CLASS_LOAD	dynamic load of a Java class
JAVA_CLASS_LOADS	a table of all dynamically-loaded classes

In addition, there are “user event” nodes, marked with a *u* icon, which may have various labels. Examples of user events are program statistics inserted from the *Probes Pane*; exception events added by the [exceptions](#) or [java_exceptions](#) pre-defined UALs; and “Data File Change” events indicating the point in time where a new data file was started.

Also, there are snapshot events, indicated by a black *S*. This will be followed by the event name, such as EXCEPTION or ROOTCAUSE_SNAPSHOT.

Trace Display Popup Menu

Select a node in the tree and use the right mouse button (MB3) to display a popup menu. This displays the operations available in the [Edit Menu](#):

- [Deselect Function In Trace Setup](#)
- [Find Function In Trace Setup](#)
- [Find Class In Trace Setup](#)
- [Find Function In Trace Events](#)
- [Find Source](#)
- [Show Associated Table](#)
- [Find Text in Trace Events](#)

In addition, the following operations from the *File Menu* are provided to operate upon APP_TRACED nodes and APP_START nodes:

- [Add Selected Process Data](#)
- [Open Associated Workspace](#)

Source Pane

The *Source Pane* displays the source file associated with the currently selected trace event in the *Event Trace Tree*. If the event doesn't have source associated with it, some other explanatory text may be shown.

Event Details Pane

The *Event Details Pane* displays additional data associated with the currently selected event in the *Event Trace Tree*. This includes data items logged on entry to or exit from methods and when a line is reached, and the time associated with each event. Complex details are organized as a tree; expand the branches of the details tree to view the data.

Call Stack Pane

The *Call Stack Pane* displays the simulated call stack for the currently selected event in the *Event Trace Tree*. This includes only methods that have been traced (and appear in the Event Trace Tree). The user should log tracebacks using *Log Traceback* in the *Probes Pane* of the *Trace Setup Dialog* to get the actual call stack.

Program Information Pane

The *Program Information Pane*, labeled "Program Info", displays information about the program(s) that were traced and the threads within the program(s). Each displayed APD file will be represented by a node describing the APD file, the program, the process, and the computer "HostID". Beneath each APD file will be nodes for the program start and end, and nodes for each of the threads started by the program.

If the log_env predefined UAL was selected in the *Workspace Tree*, the information recorded by that probe will appear in the program information pane as well.

Data Files Pane

The *Data Files Pane* displays the data files from which the trace was built.

Find Text in Trace Events Dialog

The *Find Text in Trace Events Dialog* is opened by clicking the [Find](#) button along the top of the window, or choosing [Find Text in Trace Events](#) from the [Edit Menu](#) or the [Trace Display Popup Menu](#). It is used to search for trace events that contain a given pattern. You can select a trace event in the tree to be the starting point of the search (otherwise it starts at the first event). You specify the string to search for, and can choose to consider case when matching. The next or previous event that contains the given search string will be selected, or a dialog will indicate that no matches were found. Note that you can select alternate starting nodes while the search dialog is visible to direct the search.

Buttons

Use the *Next* button to find the next occurrence of the string in an event. Use the *Previous* button to find the previous occurrence of the string in an event. Use the *Cancel* button to dismiss the search dialog. Use the *Help* button to figure out what is going on and how to get things done.

Table Dialog

The *Table Dialog* is used to display table data associated with trace events, specifically the CALL_COUNTS, LOAD_SHED, and JAVA_CLASS_LOADS events. The Table Dialog displays a description of the data at the top, the table of data in the center, and a legend at the bottom.

You can sort on a column of the table by clicking the label of that column. Click again to reverse the sort.

CALL_COUNTS Table

The Call Counts Table Dialog associated with the SYN_JAVA_CALL_COUNTS nodes is the

Call Counts Table Popup Menu

The table data contains method names and the call count for each one. Select a row in the table and use the right mouse button (MB3) to display a popup menu. This provides the same operations as are available on ENTRY and EXIT nodes in the [Event Trace Tree](#):

- [Deselect Function In Trace Setup](#)
- [Find Function In Trace Setup](#)
- [Find Class In Trace Setup](#)
- [Find Function In Trace Events](#)

In addition, the operation

- Find Class in Trace Events

is provided to search from the start of the event tree for the class name using the [Find Text in Trace Events Dialog](#).

Buttons

Use the *Dismiss* button to dismiss the dialog. Use the *Help* button to figure out what is going on and how to get things done.

JAVA_CLASS_LOADS Table

The Java Class Loads Table Dialog is associated with the JAVA_CLASS_LOADS node near the end of the event tree.

Java Class Loads Popup Menu

The table data contains class names. Select a row in the table and use the right mouse button (MB3) to display a popup menu. This provides the following operations:

- [Find Class In Trace Setup](#)
- Find Class in Trace Events

is provided to search from the start of the event tree for the class name using the [Find Text in Trace Events Dialog](#).

Buttons

Use the *Dismiss* button to dismiss the dialog. Use the *Help* button to figure out what is going on and how to get things done.

LOAD_SHED Table

The *LOAD_SHED Table* is associated with the LOAD_SHED ‘S’ node at the end of the trace events tree. This table displays information about methods for which tracing was disabled by [load shedding](#) during the previous run due to excessive overhead. Each entry shows the method that was disabled, the time at which it was disabled (to compare to other elements in the tree), and it’s *Status*: what will happen to the method the next time it is traced from this workspace. The status may be:

Don't Trace: don't try to trace it next time (default);

Load Shed: disable if it takes too much time, (as for the previous run)

Don't Shed: trace it, and don't disable it even if overhead is high.

Only methods for which the action is *Load Shed* will appear in the table will appear in this table after the next trace. Those marked *Don't Trace* will not be traced at all and so will not appear in the trace event tree at all. And those marked *Don't Shed* will always be traced, regardless of overhead.

The status for all selected rows may be changed using operations in the table popup menu, described below.

The action for an individual row may be changed by clicking on the entry in the *Status* column. This displays an option menu from which you may select the desired status.

As with other tables, you can sort on a column of the table by clicking the label of that column. Click again to reverse the sort.

LOAD_SHED Table Popup Menu

The right mouse button (MB3) displays a popup menu to operate on selected rows in the table. You can use 'Ctrl-A' to select all items, or hold down the Ctrl or Shift keys while clicking to select multiple items in the usual way.

The popup menu provides operations unique to the LOAD_SHED table:

Don't Trace Selected Functions: Use the *Don't Trace Selected Functions* menu item to change the Status of the selected functions to *Don't Trace*. All methods marked as *Don't Trace* are updated in the Trace Setup Dialog when the table dialog's *Update* button is clicked.

Don't Load Shed Selected Functions: Use the *Don't Load Shed Selected Functions* menu item to change the Status of the selected functions to *Don't Shed*.

Load Shed Selected Functions: Use the *Load Shed Selected Functions* menu item to change the Status of the selected functions back to *Load Shed*, the default behavior. The specific point at which a method may be load shed is set in the *Enable Load Shedding* option of the *Global Trace Options Dialog*.

The popup menu also provides these standard operations from the [Edit Menu](#):

- [Find Function In Trace Setup](#)
- [Find Function In Trace Events](#)

Buttons

Use the *Update* button to apply any changes made to the Status fields in the table to the corresponding functions. Use the *Cancel* button to dismiss the dialog without making any changes. Use the *Help* button to figure out what is going on and how to get things done.

Platform-Specific GUI Issues

The RootCause GUI and Different JREs

The RootCause GUI is implemented in Java. Java is supported differently on different operating systems. The RootCause installation includes a [JRE](#) (Java Runtime Environment), which is used by default when the [rootcause open](#) command is run. If you would prefer to use a Java installation other than the one shipped with RootCause you may define the environment variable `APROBE_JRE` to point to the java program, for example:

```
export APROBE_JRE=$(whence java)
```

or

```
setenv APROBE_JRE /opt/j2rel_3_101/bin/java
```

Note that this must be Java version 1.2.2 or newer.

Solaris RootCause is shipped with two [JREs](#) to ensure that the GUI will run on all versions. You will find that Java runs best on Solaris 8 or newer, where it can use the newest JRE.

AIX version 5.1 or newer is required to run the Java 2 runtime required by the RootCause GUI and other workspace-related commands that use Java.

X-emulators: (Exceed, Reflection)

We have written the RootCause GUI in Java using the Swing components. These Swing components do not work well with X windows emulators such as Hummingbird Exceed and Reflection. We are investigating this, but the GUI is best viewed from a native Unix X display.

We have seen that if you set eXceed to use an X window manager, and start the Motif window manager (mwm) or a similar window manager on the Solaris host, this works around the common problems seen with Exceed.

To do this, go to the Exceed configuration (you can get to this by right-clicking on the Exceed button in the toolbar if it's running and selecting Tools/Configuration). Next, select Screen Definition, and in the 'Screen 0' tab, set Window Manager to be "X". Click "OK" when prompted to perform a server reset which will then close all of your X windows. Open a new Exceed X window and start the Motif windows manager by executing `/usr/dt/bin/dtwm`. Then launch the

RootCause GUI from this window. GUI presentation should be improved, however there may be no window borders.

If the RootCause windows do not appear as described in the documentation when running with a Reflection X server, open the X Client Manager, and select Window Manager from the Tools menu. Select Microsoft Windows as the Default Local Window manager. Select Microsoft Windows desktop as the Window mode. Note that you must reset the Reflection X server for a change in the Window Mode to take effect, which will close all of your X applications. Hit *OK* or *Apply* to commit any changes you have made.

RootCause has been exercised under Reflection X Version 8, the version being shipped by WRQ as of 2001-06. These directions may or may not apply to other versions. Contact support@ocsystems.com for further details.

APROBE_WM_WORKAROUND Environment Variable

The APROBE_WM_WORKAROUND environment variable, when set to true, will stop the RootCause GUI from trying to set the location or size of shell windows. We have found that this eliminates a lot of problems when using an X-server such as Exceed to display the RootCause GUI. The default value is set to false.

Some GUI windows are not displayed correctly on a PC when using the Exceed X-server with a "native" window manager while logged onto a Solaris or Unix platform.

The problem is that, even though the GUI requests a window position or size, these "hints" are not always honored by the native window manager running on the PC. The result is often incorrect window size or placement.

If you are using a native window manager that does not honor these hints, you can set APROBE_WM_WORKAROUND to prohibit the RootCause GUI from requesting them.

APROBE_MONOSPACED_FONT Environment Variable

The APROBE_MONOSPACED_FONT environment variable allows the default monospaced font to be changed. We have found that using a different monospaced font eliminates some font problems when using an X-server such as Exceed to display the RootCause GUI.

The font assigned to this environment variable is passed through as a Java property that is used in UserPreferences. Unfortunately, it is difficult to determine the correct font to specify. We have had good results with the following alternate monospaced fonts:

```
APROBE_MONOSPACED_FONT=LucidaSansTypewriter-PLAIN-12
```

```
APROBE_MONOSPACED_FONT=monospaced-PLAIN12
```

Setting Background Colors

The properties in the [rootcause.properties](#) file, described in [Chapter 7, "Root-Cause Files and Environment Variables"](#), specify the background color for RootCause windows. The default is white, but you can change this to any of the Java-defined color names, or using a 6-character hexadecimal RGB value such as #FAEBD7.

Copy/Paste to/from Clipboard

The RootCause GUI is implemented in Java. Prior to JRE 1.4 the mouse-based copy and paste operations didn't work for JTextField and JTextPane classes used in RootCause. RootCause includes a version 1.4 JRE which it uses if possible, which is generally only on Solaris 8 and newer.

On earlier versions of the JRE, including the “fallback” version 1.2.2 that is also included with RootCause, there were two problems. The biggest problem was that these classes didn't have built-in support for cut/copy/paste but it could be explicitly added. We have added this explicit support for the RootCause main window's Message pane, but not for text areas like the Source pane or text fields in dialogs.

The other problem was that, even for explicitly implemented clipboard operations, the “clipboard” buffer that Java read from for Paste and wrote to for Copy was not the same as the “primary selection” buffer read/written by the default mouse-based operations.

There's nothing to be done about the first problem; unless you're using Solaris 8 you won't be able to copy or paste from text fields in RootCause. However, there is a workaround for the second problem:

There is an application called “xclipboard” that's standard on Solaris (in /usr/openwin/bin) that provides an interface between these two X Windows clipboards. So to copy something from the RootCause message window, one would:

1. Start the xclipboard application
2. Select the desired text from the RootCause message window
3. Use the "Copy" operation in the workspace [Edit Menu](#) or Messages Pane pop-up menu.
The copied text will appear in the xclipboard window.
4. Use your mouse in the normal way to copy the text from the xclipboard window.
5. Use your mouse in the normal way to paste to some other (e.g., mailer) window.

A more drastic alternative is to change your X resources to always use the same clipboard as Java does. This requires that you restart all xterms and other applications from which you might want to copy/paste, and can be cumbersome with other applications that use both kinds of clipboards such as xemacs. However, if you insist, you add the following to your X resources:

```
*.VT100.Translations:#override \  
\~Shift \~Ctrl \~Meta <Btn1Up>:select-end(CLIPBOARD)\n\  
\~Shift \~Ctrl \~Meta <Btn2Up>:insert-selection(CLIPBOARD)  
\n
```

RootCause Command Reference

The following commands are available from the command line after RootCause has been installed and the setup script in the RootCause installation directory has been executed (see [Chapter 4, "Getting Started"](#)).

RootCause and Different Shells

Different shells on Solaris have different capabilities. The following differences apply to the different shells:

sh (Bourne shell):

The `rootcause_on` and `rootcause_off` commands are not available. Instead, you must use the dot commands:

```
. rootcause_enable  
. rootcause_disable
```

ksh (Korn shell)

You may use `rootcause on` and `rootcause off` instead of `rootcause_on`, and `rootcause_off`, because `rootcause` is defined as a shell function. Note that RootCause requires that `ksh` be installed, though you need not use it as your shell. On Linux you may have to install `pdksh`.

csh (C shell)

`rootcause_on` and `rootcause_off` are aliases defined in your shell when you “source” `setup.csh`. C shell does not support shell functions, so “`rootcause on`” and “`rootcause off`” won’t work.

bash

The `setup` script and shell functions for `ksh` work for `bash` as well. However, `ksh` is still needed for `install_rootcause`, `rootcause_status`, and other scripts.

rootcause

The `rootcause` command is designed to run in a simple, intuitive manner when default file names are used. When run with no arguments, it gives version and license information. When run with `rootcause -h`, it shows the following commands, which are described in detail in this chapter.

<code>rootcause build</code>	build traces/probes in workspace.
<code>rootcause collect</code>	collect agent workspace data for analysis.
<code>rootcause config</code>	show current configuration information.
<code>rootcause decollect</code>	unpack collected workspace data for analysis.
<code>rootcause deploy</code>	package a workspace for remote deployment.
<code>rootcause format</code>	format data in workspace.
<code>rootcause log</code>	perform operations on rootcause log file.
<code>rootcause merge</code>	merge two workspaces to create a third.
<code>rootcause new</code>	create a new workspace.
<code>rootcause_off</code>	disable rootcause intercept of applications.
<code>rootcause_on</code>	enable rootcause intercept of applications.
<code>rootcause open</code>	start the RootCause GUI.
<code>rootcause register</code>	register an application with a workspace.
<code>rootcause run</code>	run any command under rootcause.
<code>rootcause status</code>	show if rootcause is enabled.
<code>rootcause xrun</code>	run a command under rootcause in a separate window.

rootcause build

The `rootcause build` command updates a RootCause workspace without opening the GUI. This is useful for maintaining workspaces as part of a script-driven product development process. The location of a workspace is provided, along with paths to all relevant programs and modules whose locations or contents may have changed. Note that a side-effect of this process may be to lose traces that no longer apply to a changed module.

Syntax:

```
rootcause build
  [-Fh] [ -x program_file | file.class | file.jar ]
  [-m module]*
  [-w] workspace.aws
```

Options:

- F** force the build even if the workspace is locked.
- h** give this command's usage
- x *program_file*** the executable program, or the `.class` or `.jar` file containing your Java application's main entry. This is the same as the argument to [Reset Program](#) in the GUI.
- m *module*** the path of a [dynamic module](#) that the program applies to. This is the same as the argument to [Reset Dynamic Module](#) in the GUI.
- w *workspace.aws*** an existing RootCause workspace.

Examples:

1. Rebuild workspace `Pi.aws` against current modules in case they've changed:

```
rootcause build Pi.aws
```
2. Update the RootCause self-analysis workspace for the current installation location:

```
rootcause build -x $APROBE/lib/probeit.jar
                 -m $APROBE/lib/libdebugInfo.so
                 -w $APROBE/arca.aws
```

rootcause collect

The `rootcause collect` command is executed on a remote computer where the RootCause Agent component has been installed to gather the RootCause data together into a single `.clct` file to be transmitted to a computer where the RootCause GUI component has been installed for subsequent decollection and analysis. It examines the rootcause registry to determine the workspace for the classes, if no workspace is specified. Multiple classes and workspaces may be specified for collection. If no arguments are supplied, the RootCause log and registry are collected.

Syntax:

```
rootcause collect
  [-AFh] [ -o clct_file ] [ -f other_file ]
  [ [-x] program_file | -c class | [-w] workspace.aws ]...
```

Options:

- A** suppress generation and collection of for native modules. This might be done to reduce the download size if you are sure the local and remote modules are identical.
- F** force overwriting of *clct_file*, if present.
- h** give this command's usage
- o *clct_file*** the collect file to create (default: *first_argument.clct*)
- f *other_file*** any other file (not directory) to be added to the *clct_file*. You can also simply copy files or directories into the workspace.
- x *program_file*** the registered program to which deployed workspace applies
- c *class*** the registered Java class to which deployed workspace applies
- w *workspace*** the workspace contents to be collected if program or class is not known

Examples:

1. The following command collects the data for the Java class `Pi` and the workspace `fred.aws` and places those two RootCause traces into the single file `myserver.clct`.

```
rootcause collect -c Pi -w fred.aws -o myserver.clct
```

rootcause config

The rootcause config command reports current configuration information. With no arguments it shows the installation directory and license information.

Syntax:

```
rootcause config [ -dhLLnRuVv ]
```

Options:

- d** give installation directory (that is, the value of \$APROBE)
- h** give this command's usage
- l** give license information
- L** give application log path (\$APROBE_LOG or default location).
- n** give product name (Console or Agent).
- R** give application log path (\$APROBE_REGISTRY or default location).
- u** give user directory (\$APROBE_HOME or default).
- v** give product version number.
- V** give product version description (default).

Examples:

1. Show the current installation information:

```
$ rootcause config
RootCause Console 2.0.5 (030405)
Installed in /app1/product/aprobe
This product is licensed to 1111 OC Systems, Inc.
This license will expire on 31-dec-2003.
```

rootcause decollect

The rootcause decollect command unpacks a **.clct** file built by the [rootcause collect](#) command. This function is also performed by the [Decollect](#) operation in the RootCause GUI (see "[Decollect Data Dialog](#)" on page 8-21).

The result of this operation is a directory tree whose root directory has suffix **.dclct**.

Syntax:

```
rootcause decollect [-F] [-o directory] clct_file
```

Options:

-F force delete of *directory*, if present
-o *directory* extract into *directory* (default: *clct_file_name.dclct*)
clct_file collect file that was built by [rootcause collect](#)

Examples:

1. Decollect the data in `myserver.clct` into `myserver.dclct`
`rootcause decollect myserver.clct`

rootcause deploy

The rootcause deploy command packages a [workspace](#) for use in a [remote \(agent\)](#) environment. This function is also performed by the [Deploy](#) operation in the RootCause GUI (see "[Deploy Dialog](#)" on page 8-19). The result of this operation is a zip file with suffix **.dply**. **Note** that this command does not verify the workspace is already built. If you're not sure, do [rootcause build](#) first.

Syntax:

```
rootcause deploy
  [-Fh] [ [-x] program_file | [-c class] [-w]workspace.aws ]
  [-l license_file] [-m module] [-o dply_file]
```

Options:

- c class** the Java class registered with the workspace you wish to deploy.
- F** force overwriting of *dply_file*, if present.
- h** give this command's usage
- l license_file**
 the agent license file to include in the deployed workspace (default \$APROBE/licenses/agent_license.dat).
- m module** is a [module](#) ([shared library](#)) for which an [ADI file](#) should be generated.
- o dply_file** the deploy file to create (default: *workspace.clct*)
- x program_file**
 the program registered with the workspace you wish to deploy.
- w workspace.aws**
 an existing, *built* RootCause workspace.

Examples:

1. Deploy workspace Pi.aws.
 rootcause deploy Pi.aws.
2. Deploy workspace forclass Factor and module libFactor.so into Factor.dply.
 rootcause deploy -c Factor -m /app/lib/libFactor.so -o
 Factor.dply

rootcause format

The `rootcause format` command runs [apformat](#) on the data collected in the specified workspace. This produces output similar to that produced by [Save As Text](#) in the RootCause GUI. By default `rootcause format` operates on the most current process. Because it formats all the data it can take a while for large amounts of data. You can use the `-O` option in conjunction with the `apformat` “-n” option to limit it to specific APD files, as shown in Example 3 below.

Syntax:

```
rootcause format
  [-h] [-p PID] [-O "options"] [-t tmpdir]
  [-w] workspace.aws
```

Options:

- h** give this command's usage
- l** list the [APD rings](#) ([Process Data Sets](#)) in the workspace, but don't format anything. The newest data set is listed first.
- r** raw: just run `apformat` directly on the APD file (with options specified using **-O**) rather than using the workspace's formatting script.
- p PID** format data for the process given by [PID](#)
- O "options"**
pass *options* to the `apformat` command. The options must be in quotes, and quotes in the options themselves must be preceded by a backslash.
- t tmpdir** specifies the directory where intermediate files are to be produced. These can get very large--up to 10 times the size of the APD files depending on the formatting--and this can be used to avoid disk-space restrictions where the workspace resides.
- w workspace.aws**
the RootCause workspace containing the data to be formatted.

Examples:

1. Format the newest data set in `Pi.aws` into the file `Pi.txt`.

```
rootcause format Pi.aws > Pi.txt
```

2. List the [Process Data Set](#) in workspace Pi.aws.

```
$ rootcause format -l Pi.aws  
/work/Pi.aws/Pi.class.apd.11991/Pi.class.apd  
/work/Pi.aws/Pi.class.apd.11785/Pi.class.apd
```

3. Run [apformat](#) directly on the newest data file for process 11785 in Pi.aws.

```
rootcause format -r -O "-n 0" -p 11785 Pi.aws
```

rootcause log

The `rootcause log` command provides information about the RootCause Log, and allows its maximum size to be changed.

Syntax:

```
rootcause log [ -hlnsFZ | -s size ]
```

Options:

- F** force `-s` size or `-Z` operation without confirmation
- h** give command-line help
- l** list log file contents to standard output
- n** list the log file name to standard output
- s** list the log file size to standard output
- s** *size* set the maximum size of the log to *size* bytes (size > 1000)
- Z** clear the contents of the log file

Examples:

1. Write the contents of the log to standard output:

```
rootcause log
```
2. Set the size of the log to 20000 bytes:

```
rootcause log -s 20000
```

rootcause merge

The `rootcause merge` command merges two [workspaces](#) to create a new, third workspace. It works by copying the first *primary* workspace to the third *result* workspace, then adding compatible traces and UALs from the second *secondary* workspace. A [module](#) must exist in both the *primary* and *secondary* workspaces in order that traces for that module appear in the *result* workspace.

There is no GUI operation equivalent to `rootcause merge`. You can use it in conjunction with the GUI by:

- Using *Workspace->Close* to close your current workspace
- Applying `rootcause merge` from the command line
- Using *Workspace->Open* on the result workspace.

Note: The [rootcause build](#) and [rootcause register](#) operations must be applied to the *result* workspace before the result workspace can be used to trace an application.

Syntax:

```
rootcause merge [-Fh] primary.aws secondary.aws result.aws
```

Options:

- F** force *result.aws* to be overwritten if it exists
- h** give command-line help

primary.aws

The primary workspace, on which the result workspace is based.

secondary.aws

The secondary workspace, from which additional traces and UALS are added to the result workspace.

result.aws

The new workspace that is created.

Examples:

1. Merge traces in *PiDetails.aws* into *Pi.aws* to produce *PiPlus.aws* and make *PiPlus.aws* the new workspace for tracing *Pi*.

```
rootcause merge PiDetails.aws Pi.aws PiPlus.aws
rootcause build -w PiPlus.aws
rootcause register -c Pi -w PiPlus.aws
```

rootcause new

The `rootcause new` command creates a new [workspace](#). Generally this is done through the RootCause GUI using the [New](#) menu item or [Open Associated Workspace](#); (see "[New Workspace Dialog](#)" on page 8-9). The result of this operation is the named workspace, initialized to do default tracing. If the `-r` option is used, the workspace is also registered with the specified program or Java class.

Syntax:

```
rootcause new  
    [-Fhr][-c class] -x program_file [-w]workspace.aws ]
```

Options:

- `-c class` the Java class registered with the workspace you wish to deploy.
- `-F` force overwriting of *workspace.aws* if it exists.
- `-h` give this command's usage
- `-r` register the new workspace with the specified program or Java class
- `-x program_file`
 the executable program or Java `.class` or `.jar` file the workspace will be used to trace (as on the [rootcause open](#) command).
- `-w workspace.aws`
 the new workspace to be created.

Examples:

1. Create and register a new workspace for `Pi.class`.

```
rootcause new -r -x Pi.class -c Pi -w Pi.aws.
```

rootcause_off

Use the `rootcause_off` command to disable rootcause interception of processes on your machine.

Syntax:

```
rootcause_off
```

rootcause_on

Use the `rootcause_on` command to start the inspection and interception of processes on your machine to determine if they should be traced with rootcause.

Syntax:

```
rootcause_on
```

rootcause open

The `rootcause open` command starts the RootCause GUI. If the application class specified on the command line is registered, the GUI will automatically set the workspace from the registry entry for the application. If the application is not registered, the GUI will prompt for a new workspace name and register the application. If no arguments are specified, the current RootCause Log file is opened.

Syntax:

```
rootcause open
  [[-x] program_file]
  [-c classname] [[-w] workspace.aws]
  [ [-d] dir.dclct | [-z] file.clct ]
```

Options:

program_file

the executable program file, or the the `.class` or `.jar` file containing your Java application's main entry

classname

the main class name. This is required if *classname* is not the same as *file*

workspace.aws

a new or existing RootCause workspace

dir.dclct

a directory created by the RootCause *Decollect* operation

file.clct

a file created by the `rootcause collect` command

Examples:

1. Start the RootCause GUI and examine the RootCause Log file in a Trace Display window.

```
rootcause open
```

2. Start the RootCause GUI to open new or existing workspace `converter.aws`.

```
rootcause open converter.aws
```

3. Start the RootCause GUI to open a new or existing workspace for main class `Pi` compiled into file `Pi.class`.

```
rootcause open Pi.class
```

4. Start the RootCause GUI to open a new or existing workspace for main class `com.ocsystems.probeit.Main` compiled into file `probeit.jar`.

```
rootcause open probeit.jar -c com.ocsystems.probeit.Main
```

5. Start the RootCause GUI to unpack (decollect) the collected rootcause data in `pi_demo.clct`.

```
rootcause open pi_demo.clct
```

rootcause register

The `rootcause register` command provides the interface to the RootCause registry. The GUI will allow you to add or delete the current workspace from the registry, but you must use the `register` command to otherwise manipulate the registry. It is likely that, over time, more GUI support will be added to manipulate the registry, but on computers where only the RootCause Agent is installed, there is no GUI and the `register` command must be used.

Syntax:

```
rootcause register [subcommand ] options [deploy_file]
```

Description:

subcommand. The *subcommand* flag designates the operation to be performed:

- a** add a new entry in the registry (default)
- d** delete an entry from the registry
- h** give command help
- k** return 0 iff specified args are already registered & enabled
- l** lists all registry contents
- lr** list registry name only
- lw** list workspace name only
- lx** list only registered JVM or class only
- s debug** enable/disable debug mode with **-e on/off** (off by default)
- s verbose** enable/disable verbose mode with **-e on/off** (on by default)
With verbose mode on, all processes are recorded in the log;
with verbose off, only traced applications are recorded.
- Z** clear entire registry contents, including **-s** settings, returning them to their default values.

options. Options further qualifying the above are:

- c classname** probe Java commands naming main class *classname*
- e on | off** off specifies 'disabled' (default: on)
- F** force without confirmation
- j dir** dir is root of JRE containing java exe to probe
- m file** file is a module required for *deploy_file* consistency checking

-r *file* file is registry file to use
-w *dir.aws* file is workspace to use
-x *file* file is executable to probe
deploy_file is a `.dply` file to unpack into a registered workspace

Examples:

1. List the registry name and contents:
`rootcause register -l`
2. Delete the registry entry for class `Pi`:
`rootcause register -d -c Pi`
3. Turn off recording of all processes in the RootCause log:
`rootcause register -s verbose -e off`
4. The following command will do the following all in one step:
 - register the program
 - create the workspace (if it does not exist)
 - deploy the trace into the workspace

This would be the typical command used on a remote computer where only the RootCause Agent component was installed in order to implement a `.dply` file generated by the RootCause GUI component. After this command is issued, you would merely execute `rootcause_on` in the context of the shell and run the application.

```
rootcause register jfrob.dply
```

rootcause run

Use `rootcause run` before your command to cause it to be [run with rootcause on](#), independent of the current [rootcause status](#). The command specified will be run in the current window exactly as if it were not preceded by `rootcause run`. This is equivalent to

```
rootcause_on
command
rootcause_off
```

Syntax:

```
rootcause run command
```

Options:

command any shell command

Example:

1. Run the Pi application with rootcause on:

```
rootcause run java -cp $APROBE/demo/RootCause/Java Pi
```

rootcause xrun

Identical to [rootcause run](#), but the *command* is run in a separate window. This is used by the [Run](#) button in the RootCause GUI.

Syntax:

```
rootcause xrun command
```

rootcause status

Use the `rootcause status` command to show whether rootcause tracing is currently enabled or disabled.

Syntax:

```
rootcause status
```

This chapter contains discussions of various RootCause topics that may be of interest to you, the RootCause user.

RootCause and Efficiency Concerns

RootCause should preferably be installed on a local file system. It will work if it is mounted on a remote file system, but this may also impact performance.

The RootCause workspace should be created on a file system that is local to the machine on which the traced process will be run. The data logged by RootCause is written to the workspace. If the workspace is remote, then the logged data will have to be transmitted across the network, increasing the overhead of logging as much as tenfold. See also, "[RootCause Data Management](#)" on page 3-4.

RootCause adds probes to the application in memory. These probes are optimized machine code, so while they are fast, they must of course add overhead to the execution of the application. RootCause only “patches” the traced functions and methods. For Java, RootCause inserts byte code to only trace the methods of interest, not all methods.

Furthermore, RootCause tracing applies automatic “[load shedding](#)” to automatically turn off tracing of functions that are introducing high trace overhead. Such functions can then be removed from the trace specification by the user in the next run. Using this mechanism and by adjusting the load shedding level, one can

quickly get to an acceptable level of overhead. See ["RootCause Overhead Management" on page 3-7](#).

Typically, we have seen that one can add a 5% load and still get a useful trace. In general, you will have to iterate to define a good trace that adds a reasonable load so the application can still run in the operational environment. Note that RootCause supports this workflow, by allowing one to choose (and remove) trace items from the viewer to speed the removal of “noise” routines (noise routines are those that add little value to the trace).

Note that a program being probed by RootCause, will take somewhat longer to start. Typically, a few extra seconds are required for a RootCause session on an application. This minimal overhead is incurred because RootCause does as much as possible up-front, to reduce the runtime penalty later.

Solaris SETUID, and Security Concerns

This section briefly describes how RootCause / Aprobe can be used with certain “secure” applications on Solaris. These mechanisms are not yet provided for other platforms; contact OC Systems for more information.

The Solaris operating system provides a secure environment for debugging and running your applications. RootCause and Aprobe do not interfere with this mechanism but extend it to work safely in a number of environments that require it.

For the purposes of this document, a secure application is one that has the setuid bit set. We discuss how the Solaris security mechanism works with these applications and how Aprobe and RootCause provide their own extensions to the Solaris security protections to allow you to safely run probes on these applications without compromising system security.

Note that this document does not discuss applications with the setgid (group) bit set. At the time of writing, Aprobe and RootCause do not support running such applications.

Avoiding Solaris Warnings

Even if you do not wish to probe secure applications, you may want to place `libapaudit.so` in the secure location anyway to eliminate error messages. If you do not do this and try to run RootCause on an application that has the SETUID bit set, you will get an error message something like:

```
ld.so.1: mail: warning: /opt/RootCause/lib/libapaudit.so:
open failed: illegal insecure pathname
ld.so.1: mail: fatal: /opt/RootCause/lib/libapaudit.so: audit
initialization failure: disabled.
```

Although these look like fatal errors, the application ran without error, and it was only the loading of `libapaudit.so` that failed.

Placing `libapaudit.so` in the secure location as described below will allow `libapaudit.so` to load for SETUID applications like `/usr/bin/mail`, so it can determine whether to probe the new process or not.

Note that just placing `libapaudit.so` in the secure location does *not* allow one to actually probe the SETUID application unless one is running as the effective user.

The secure path for dynamically-loaded libraries is different on each version of Solaris. This logic is encapsulated in a script, `rootcause_libpath`.

The simplest usage is:

1. Log on as root so you have write access to `/usr/lib` and its subdirectories.

2. Set up for using RootCause, e.g.,

```
. /opt/RootCause/setup
```

(see ["The Setup Script" on page 4-1](#)).

3. Run the command:

```
rootcause_libpath -c
```

This will copy the appropriate library to the secure locations. These locations are under `/usr/lib`, so you must be super-user. The script assumes that you are set up for RootCause, so you must run the RootCause setup script first. You should see output like:

```
/usr/lib/libapaudit.so correctly installed.  
/usr/lib/secure/libapaudit.so correctly installed.  
/usr/lib/64/libapaudit.so correctly installed.  
/usr/lib/secure/64/libapaudit.so correctly installed.
```

4. Log off root on this machine.
5. You will need to do this on each machine on which you use RootCause.
6. After doing this, you will need to do `rootcause_off`, then `rootcause_on` again to pick up the new values.

Description of Solaris Security

This section briefly describes the Solaris security measures that are appropriate for RootCause / Aprobe. It should be noted that each version of Solaris has its own subtle variations on this. All examples given are for Solaris 8 and over although, with the exception of Solaris 2.5.1, RootCause and Aprobe can be expected to behave identically on older versions as far as security goes. (Solaris 2.5.1 has overly tight restrictions that were corrected in later versions).

The first concept that must be understood is that every executable run has two users associated with it at runtime. The first is the “real” user, the logged in user -

the user shown when you use the command “id”. The second is the “effective” user which really governs the permissions you have during runtime.

(One important point is that if the real user is root, all security mechanisms are effectively disabled because they are moot. One practical result of this is that you may use Aprobe on any application if you are logged in as root).

Normally the real and the effective user are the same. If, however, the setuid bit is set on an application, the operating system changes the effective user to match the owner of that application. Most commonly this is the root user and is done to give a regular user temporary access to a limited set of secure resources.

Let’s take the “/usr/bin/at” command as an example. The output from “ls -l” might look like this:

```
-rwsr-xr-x  1 root sys  37876 Jul 10  2000 /usr/bin/at
```

Note that instead of an ‘x’ where we would expect the owner’s executable bit, we see a ‘s’. This means that the application will run with the effective user root, with all the permissions that that allows.

What would happen if we were allowed to attach a debugger to this application? Suddenly we would be able to cause the application to execute arbitrary instructions as if it were root! To prevent this, the operating system will prevent the debugger interface being used in such a situation. (Again, if you are actually logged in as root, you will be allowed access).

Another aspect of security for these applications is where they load their libraries from. Obviously the application can have a set of specific libraries linked in and these can be safely loaded. But the runtime linker also provides some capabilities to add arbitrary shared libraries in using the LD_PRELOAD and LD_AUDIT runtime linker environment variables. Once again it would be a security risk if any library could be specified, so the operating system only allows libraries in “secure” paths to be loaded by these environment variables.

Impact of Security Measures on Aprobe

When we run the “aprobe” command on an executable, we start out life as a debugger, patching in the probes that we’ve specified. Once this is done, the “aprobe” executable detaches from the application and goes away. As was mentioned above, Solaris will not allow the use of the debugger interface on a secure

application. Aprobe will specifically check for this so it can give a more friendly warning if you try to run it:

```
$ aprobe /usr/bin/at
(E) /usr/bin/at
This file is owned by root and has the setuid bit set.
You need to use the secure version of aprobe (saprobe) to run this
application under Aprobe. Please see the section on secure applications
in the Aprobe user's guide.
```

As this error describes, there is a secure version of Aprobe that allows us to run on these applications. In fact, there are three ways we could run this application:

1. Log in as root. As was mentioned above, security restrictions are moot for the root user and so Aprobe will run fine.
2. If you could rebuild or relink the application, you could link in the libdal.so file that allows an executable to patch itself. The use of this is outside the boundaries of this document but you can find more details in the Aprobe user's guide.
3. Use the secure version of Aprobe mentioned above - **saprobe**. The secure version itself has the setuid bit set so that it runs as root and can attach to the application.

It doesn't take much thought to realize that option (2), if implemented blindly, could leave a big security hole in your application. But, of course, it isn't implemented blindly. When you run saprobe on an application, the application must be listed in \$APROBE/lib/secure_applications. This file is created so that it is only writable by root and we check this is still the case at runtime before allowing its use. Let's see what happens when we try to run without an entry for it:

```
$ saprobe /usr/bin/at
(W) /usr/bin/at
You are running a secure application but the secure_applications file
did not contain an entry for it.
(F) Aprobe will not run this application due to security restrictions.
Please see the section on secure applications in the Aprobe user's guide.
```

The second level of checking is that the files loaded by Aprobe - the runtime libraries and the UALs - must all be owned by root and not writable by anyone else. Additionally, for all UALs except the default system_ual, an entry for them must exist in the secure_applications file under that application. If it doesn't:

```
saprobe -u trace /usr/bin/at
(W) "/appl/aprobeinst/fred/aprobe_sun_50/ual_lib/trace.ual":
This ual is not valid for your secure application. It must be listed in
the secure_applications file under this application.
(F) Aprobe will not run this application due to security restrictions.
Please see the section on secure applications in the Aprobe user's guide.
```

The format of the `secure_applications` file is defined in its header. However, it is pretty trivial. For each application we allow we have an “APPLICATION” keyword followed by any number of “FILE” keywords. Another APPLICATION keyword automatically ends the list of allowed files. For instance:

```
APPLICATION /usr/bin/at
FILE /appl/aprobe/inst/fred/aprobe_sun_50/ual_lib/trace.ual
FILE /opt/product/probes/myprobe.ual
APPLICATION /usr/bin/another_app ...
```

Impact of Security Measures on RootCause

RootCause builds on top of Aprobe and so has the same protections described above. However, the RootCause intercept mechanism is based on the `LD_AUDIT` environment variable and must be managed appropriately.

By default, if you set `LD_AUDIT` to a specific path, Solaris will not load that audit library when the application is run. Annoyingly, later versions of Solaris give a misleading error message about this being a fatal condition which it isn't!

If, however, the audit library is in a secure location and the `LD_AUDIT` environment variable is appropriately set, it will be loaded by the runtime linker. The path to that library varies between versions of the O/S but, on Solaris 8 and higher, is `/usr/lib/secure`.

So, to allow RootCause to intercept secure applications, the audit library is placed within here. In order that this does not create a security risk in itself, RootCause ensures that it will only run an application under RootCause if the workspace's script file is secure. If it isn't, you'll get an error message and the application will be run without RootCause.

By this mechanism, we safely control access to the scripts that will execute Aprobe and trigger the protections that Aprobe introduces.

Using the Secure Version of RootCause / Aprobe

The first step that must be taken is to provide appropriate ownership, permissions and location of certain RootCause files. A normal installation of RootCause does not have a secure version of Aprobe, it doesn't locate the audit libraries in secure paths and it may not have appropriate ownership of runtime libraries and UALs.

To create a secure environment, you must log in as root and run the `rootcause_libpath` script. This takes a number of parameters and must be run on each machines on which you wish to use the secure version of RootCause.

There are two main parts to this:

1. Creation of the secure Aprobe files. This must be performed once for a given installation of RootCause / Aprobe. In many networks it must be done on the machine that the installation is directly mounted on (e.g. many NFS mounted filesystems do not allow root write access from across the network). The command to update the installation is

```
rootcause_libpath -s
```

This is described in more detail in ["Avoiding Solaris Warnings" on page 10-3](#).

2. Creation of the secure RootCause files. This must be performed once on each machine you wish to intercept secure applications on. To command to do this is

```
rootcause_libpath -c
```

Note that you can combine this and the “-s” option where appropriate.

A secondary step for RootCause is to define the workspace as secure. When creating a workspace, check the “Secure Application” checkbox to mark the workspace as secure. This will create runtime scripts that invoke the secure version of Aprobe. If, at a later time, you wish to change the security property of the workspace, you can change it in the Aprobe options tab of the RootCause options dialog (accessed from the Setup menu).

Note that if you build a secure workspace for a non-secure application or vice-versa, you will get error messages at runtime.

64 bit applications

64 bit applications are not yet supported by RootCause. If you require this support, please let us know.

Logging Controls

One of the most fundamental features of RootCause is a robust and fast logging mechanism, both for persistent and wraparound data collection.

RootCause chooses sane defaults for logging, but you may want to change them. There are several main user-selectable options for logging application data provided in the [RootCause Options Dialog](#).

See "[RootCause Data Management](#)" on page 3-4 for more information.

Multiple Application Tracing

Each application puts its trace data into an application specific workspace. This mapping of application to workspace is defined in the registry.

When viewing trace data, RootCause can add trace data from other applications/workspaces, so that you can view a fully integrated process trace. The traces are automatically ordered so there is a coherent time line for all traced applications.

RootCause collects data into separate files to eliminate contention for a single logging buffer. For example, if you are tracing 10 processes and all 10 are trying to write to the same buffer, then there will be much contention for that buffer and performance would suffer. RootCause solves this problem by logging the data into independent application specific workspaces and then combining the traces in the GUI viewer.

A trace is merged with an existing trace using the [Add Selected Process Data](#) operation in the [Trace Display Popup Menu](#) of the [Trace Display](#) window. You can then use [Save As XML](#) or [Save As Text](#) to save this merged trace for future examination.

This is illustrated by the Advanced demo delivered with RootCause in \$APROBE/demo/RootCause/Advanced. See the README.html file in that directory for a detailed description of that application, the separate Java and C++ portions, and the merging of combined traces.

The ability to view a single time line trace of multiple processes (even on SMP computers) is a very powerful feature of RootCause.

Multiple Executions of a Single Application

It is not uncommon in production environments for a single application to have multiple processes executing simultaneously. RootCause handles this by tracing each process independently.

As mentioned previously, each [application](#) has a [workspace](#). In the workspace there are a number of sets of [Process Data Sets](#).

RootCause automatically reuses the oldest of these process data sets upon each new invocation of the registered application. The number of process data sets to keep is specified with “[Keep logged data for N previous processes](#)” in the [RootCause Options Dialog](#).

So if you wish to trace a total of 10 simultaneous executions of your application, you will tell RootCause to create *at least* 10 process data sets in the workspace. Note that this mechanism can also be used to save serial executions of a process too. For example, if you would like to trace the last 4 executions of the registered application, tell RootCause to keep 4 previous processes.

See “[RootCause Data Management](#)” on page 3-4 for more information.

Libraries with No Debug Information

The RootCause Console GUI takes advantage of Aprobe's APC translator to provide function prototype information for C object modules in [shadow header files](#).

A shadow header file is a legal C header file, containing C type and function prototype definitions and C preprocessor directives (such as `#include`). The information in this file supplements the information in a compiled object module of the same name, resulting in more useful traces and custom probes.

When you click on the name of a compiled module, say `"libc.so"`, in the [Trace Setup Dialog](#), this causes that module to be opened and searched for debug information provided by the compiler. Then, a *shadow header file* corresponding to that module--in this case, `"libc.so.h"`-- is searched for, and if found, the information found there correlated to the symbols read from the module. This results in otherwise "unknown" functions being grouped according to the header file from which they are read, and having parameter type (and often name) information.

Shadow header files are searched for in a "shadow" subdirectory of the [.rootcause Directory](#) (e.g., `~/rootcause/shadow/libm.so.h`), and if not found there, in `$APROBE/shadow`.

OC Systems provides only one or two sample shadow header files on each platform. You're encouraged to add your own, and to contact OC Systems if you need help developing a header file for a particular library. Note that you don't have to provide all the prototypes in the library, only those you need. Conversely, if there are a few extras that aren't in the shadowed library that's okay, too -- they'll be ignored.

The easiest way to create such a file is simply to add `#include` preprocessor directives for existing C header files provided with your system or compiler. Note that these must be C header files ending in `.h`, *not* C++ header files. These are preprocessed using the same environment (include path and preprocessor definitions) as the [APC](#) files, but you can edit the files and add your own `#define` directives as necessary.

Your Application and Different JREs

If you've defined traces for a Java class in a workspace, but if after running the application under RootCause, the RootCause Log shows only an APP_STARTED (but not an APP_TRACED) event for the java program, this indicates that it wasn't recognized as Java. There could be several reasons for this:

1. The version of Java you're running isn't supported. Check the program name in the rootcause log entry against the supported JREs identified in [“System Requirements” on page 2-2](#).
2. The JRE hierarchy in which RootCause looks for files was unusual, so RootCause could not find the necessary files. In this case you can explicitly register the JRE with the java executable using the [rootcause register](#) command. This is done automatically when applying a workspace to a class in the GUI, but the JRE in the execution environment may be different.
3. The program which is running your class isn't “Java”, but some other program which loads a Java plug-in or DLL. See [“Using RootCause on an Application with an Embedded JVM” on page 10-12](#).

Using RootCause on an Application with an Embedded JVM

RootCause currently supports probing applications that process Java by using the Sun version 1.2, 1.3 or 1.4 Java runtime (“libjvm.so”) library. To do this:

- You will need a license for both RootCause for C++ and for RootCause for Java; contact OC Systems if you have questions about this.
- Set up a workspace for the application with the embedded JVM.
- Use “Add Dynamic Module” to add a Java class you wish to trace.
- When prompted, specify the full path to the `libjvm.so` library that the program uses.
- Set up a trace and run as usual.

Tracing Java and C++ In One Program

RootCause is designed to support both Java and compiled-language probes and traces in a single application. To do this, you will need a license for both RootCause for Java and RootCause for C++; contact OC Systems if you have questions about this. The RootCause GUI itself is an example of mixing Java and C in an application. It is implemented in Java, but has significant portions of its functionality implemented in C, which is dynamically loaded by Java. To see the Java/C interaction in a trace, one would:

1. Open a Java Workspace for the Java main class of the application.
2. Use *Workspace->Add Dynamic Module* to specify the dynamic C/C++ library that will be loaded.
3. Click Setup to show the Java classes and dynamically loaded module, and define your traces as usual.

Another common scenario is when a C++ application creates another process to act as its GUI, and communicates with it by sockets. In this case, one creates separate workspaces for the compiled and Java parts of the application, and merges the results, as described in “Multiple Application Tracing” on page 10-9.

RootCause J2EE Support

RootCause will work with any J2EE-compliant Enterprise Java Application Server that uses a standard JRE from Sun (version 1.2 or higher). This includes Sun iPlanet 6.5 and AS7, BEA WebLogic 5.1, 6.1 and 7, and JBOSS 3. It will also work with standalone Web Servers such as TomCat.

RootCause can trace an Application Server that is run as a standalone Java JVM (using the java executable) or it can trace a JVM that is embedded within a native executable.

If the Application Server runs as a standalone Java JVM, you can create a workspace just like any other Java application. Make sure RootCause is enabled in the shell or environment you are running the Application Server JVM. Run the Application Server, and find the Java APP_START event in the [Trace Display](#) window.

Note: you may need to increase the application server's Java heap size to accommodate RootCause tracing overhead; check your app server documentation.

In the [New Workspace Dialog](#), there is an option for "J2EE Server Directory". Enter the directory where deployable Enterprise Java Bean (EJB) and Servlet classes and jars reside. RootCause will automatically add EJB and Servlet classes and jars that are specified in any J2EE compliant XML deployment descriptors.

Once a Java workspace has been created and opened, the J2EE Modules directory can be changed to another location, or the current directory can be searched again for updated or new J2EE applications. This can be done using [Update J2EE Modules](#) in the [Workspace Menu](#).

If the Application Server runs embedded within a native executable, you can create a workspace for the native executable, and then add the libjvm library as a dynamic module. First create a workspace for the executable that runs the Application Server as you would for any other. Then open the Trace Setup window.

An Application Server might run an embedded JVM, but already have libjvm library loaded as a dynamic module. If this is the case, the libjvm library will show up in the list of loaded libraries in the [Trace Setup Dialog](#).

If libjvm does not appear as a statically-loaded module in Trace Setup, you must find the server version of the libjvm library (libjvm.so on Solaris, libjvm.dll on Windows). Once this module has been found, it can be added using [Add Dynamic Module](#) in the [Workspace Menu](#).

Once the `libjvm` module is shown in the Trace Setup window, you can complete the J2EE configuration from the main workspace window using [Update J2EE Modules](#).

RootCause Shipped as Part of Your Application

RootCause is designed to solve problems from a single occurrence while simultaneously reducing support costs. While you can wait until a user reports a problem and then use RootCause to debug it, it is an intended use of RootCause that you include it as part of your application, so your application is always logging trace data. Whenever a user encounters a problem, they merely send you the RootCause [collect](#) file, and the root cause analysis of the problem is performed from that file. This greatly simplifies the reporting and debugging of problems. In some cases, for particularly difficult problems, you may have to send a more focused trace to the user site to complete the analysis of the problem, but the RootCause workflow is optimized to do this.

If you plan to include RootCause as part of your shipped application, we suggest that you contact OC Systems support to enter into a discussion with one of our technical staff. It is not difficult, but we can discuss various issues with you to save time and effort.

In addition to the built-in traces and actions available in the RootCause GUI, RootCause also supports inserting arbitrary “probes” into an application. This allows for custom statistics-gathering, or even modifying the flow of the program.

The custom probes are themselves written in Java, and are activated by specifying them in a [deployment descriptor file](#) which is an [XML](#) file with the suffix `.xmj`.

This chapter describes how one writes these files through a graduated series of examples. The source text for these examples is found on-line in the directory:

`$APROBE/demo/RootCause/Java/Custom.`

A Simple Example

Let’s start with a simple example of how to do this, using the same Pi demo supplied with RootCause that was used in [Chapter 5, "RootCause Demo"](#).

Write the Probe In Java

1. Create a file called [MyFirstProbe.java](#) as shown below.

This probe will print messages whenever a method to which it is applied gets called.

```
import com.ocsystems.aprobe.*;

public class MyFirstProbe extends ProbeMethod
{
    public boolean onEntry( Object[] parameters )
    {
        System.out.println( "Hello world (from my probe)!" );

        for( int i=parameters.length-1; i>=0; i-- )
        {
            System.out.println(
                "Parameter # " + i + " is " + parameters[i] );
        }
        return true;
    }

    public Object onExit( Object returnValue )
    {
        System.out.println( "Goodbye world (from my probe)!" );
        System.out.println( "Return value was: " + returnValue );
        return returnValue;
    }

    public void onExit() // no return value
    {
        System.out.println( "Goodbye world (from my probe)!" );
    }

    public void onExceptionExit( Throwable e )
    {
        System.out.println(
            "Goodbye world (from my probe), due to exception:" );
        System.out.println( e );
    }
}
```

MyFirstProbe.java

Note that although all this probe does is make some calls to print via `System.out.println()`, probes may contain arbitrary Java code, to do whatever you want.

You can create any of your own probes to be applied to methods simply by extending the class `com.ocsystems.aprobe.ProbeMethod`, and overriding the `onEntry()`, `onExit()` and `onExceptionExit()` methods, as shown

above. As you can see, your probe has access to the parameters and return value of the method it is probing.

2. Then compile this `MyFirstProbe.java`. You must specify to the java compiler where it can find the classes in the package `com.ocsystems.aprobe`. Do this with the command

```
javac -classpath $APROBE/lib/aprobe.jar MyFirstProbe.java
```

This will create `MyFirstProbe.class`.

3. Copy the `.class` file into the workspace directory of the workspace you wish to use. Your workspace from the Pi example from previous chapters would be an appropriate place.

Write an XMJ File

Now that we have written a probe, how do we specify where this probe is to apply? This is done with an *XMJ file*, or *deployment descriptor file*, as follows.

4. In the workspace directory, create another file, called `MyFirstProbe.xmlj`, with the following contents:

```
<probe_deployment>
  <probe class="MyFirstProbe">
    <target value="Pi::main"/>
  </probe>
</probe_deployment>
```

MyFirstProbe.xmlj

This indicates that the probe contained in the class `MyFirstProbe` should be applied to the method `main()` in target class `Pi`. You could pick any method in any class that interested you.

Update the Workspace

5. Copy the `.xmlj` and `.class` files into the `Pi.aws` directory.

The workspace directory is added the the classpath when you run with Root-Cause, so you're ready to run if you have copied your `.class` and `.xmlj` files to the workspace directory.

NOTE: If you put your custom probes in a JAR file, you'll need to rebuild the workspace to cause that to be added to the classpath (or you can add it explicitly

to your 'java' command if that's convenient. To pick up a JAR file in the workspace, you can rebuild the workspace from the command-line with:

```
rootcause build Pi.aws
```

If you have the RootCause console open, you can just click the [Build](#) button.

Run With RootCause

6. Run the Pi program under RootCause. You may do this either by pressing the Run button in the RootCause GUI, or by turning RootCause on and running the program from the command line as described in ["Trace With RootCause" on page 5-10](#).

You will see the output from MyFirstProbe as `Pi.main()` is executed.

Applying One Probe to Many Methods

You may wish to apply a probe to many different methods. There are several ways to do this with the .xmj deployment descriptor file, without changing the Java at all: wildcards, target lists, and using multiple .xmj files.

Wildcards

You can use a wildcard string as the target value in your .xmj file to accomplish this, as shown below:

```
<probe_deployment>
  <probe class="MyFirstProbe">
    <target value="*:*:*"/>
  </probe>
</probe_deployment>
```

WildcardExample.xmj

Using [WildcardExample.xmj](#) as the deployment descriptor file will cause every method to have MyFirstProbe applied to it.

The class, the method, or both can be a wildcard. All of the following are valid:

- Probe all methods in all classes, as shown above:

```
<target value="*:*:*"/>
```

- Probe all methods in a given class:

```
<target value="Classname:*"/>
```

- Probe all methods named “foo” in any class:

```
<target value="*::foo"/>
```

Lists

You can define a list of targets, and reference that list as the target of a probe, as shown in [ListExample.xmj](#) below:

```
<probe_deployment>
  <target_list_definition name="ListExampleList">
    <list_target value="Pi::calc_pi"/>
    <list_target value="Pi::main"/>
  </target_list_definition >

  <probe class="MyFirstProbe">
    <target_list name="ListExampleList"/>
  </probe>
</probe_deployment>
```

ListExample.xmj

This deployment descriptor file shows how to construct a list. This example would apply `MyFirstProbe` to two methods, `main()` and `calc_pi()`.

Multiple XMJ Files

All `.xmj` files in the workspace directory are automatically detected and applied, so you could simply make a copy of one file, change the target name, and you'll pick up both target methods on the next run. Conversely, any `.xmj` files you do not wish to use must be removed from the workspace.

Using Method IDs

If you ran the introductory example, and applied the single probe to a number of different target methods, you may have noticed that the output can be a bit hard to figure out, because all of the methods being probed are producing the same output. However, there is built-in support to identify different target methods. Each one has a unique *method ID*, and your `ProbeMethod` knows the ID of the target method upon which it is acting.

In addition, the class `com.ocsystems.aprobe.SymbolTable` contains a variety of utility functions for manipulating method IDs. The most immediately useful such is `getPrintableMethodName()`. It converts a method ID to a `String` containing the name of the method. The method ID itself is retrieved by the

invoking the probe's own `getMethodId()` method, which is inherited from the class `com.ocsystems.aprobe.ProbeMethod`.

This next probe, [Example2.java](#), uses `com.ocsystems.aprobe.SymbolTable.getPrintableMethodName()` and a probe's method ID to make very clear the program flow in the target:

```
import com.ocsystems.aprobe.*;

public class Example2 extends ProbeMethod
{
    private String methodName = "(unknown)";

    private void printEvent( String event )
    {
        System.out.println(
            event + " (in method " + methodName + ")" );
    }

    public boolean onEntry( Object[] parameters )
    {
        methodName =
            SymbolTable.getPrintableMethodName( getMethodId() );
        printEvent( "Method Entry" );
        return true;
    }

    public Object onExit( Object returnValue )
    {
        printEvent( "Method Exit" );
        return returnValue;
    }

    public void onExit() // no return value
    {
        printEvent( "Method Exit" );
    }

    public void onExceptionExit( Throwable e )
    {
        printEvent( "Method Exit via exception" );
    }
}
```

Example2.java

This example also shows that your probe class is just another class. You can define any of your own fields and methods in it, and use them as you would in any other class, just as we did here with `methodName` and `printEvent()`.

Logging Data from Java

So far, we have sent all output from our probes to `System.out`. This mixes up the probe's output with the application's output, and makes post-runtime analysis more difficult. As you've seen in earlier chapters, `RootCause` normally [logs](#) data at runtime to be examined later. You can use the same mechanism in your custom probes. This is used most easily through the class `com.ocsystems.aprobe.Logger`.

The following probe uses the method `log()` in that class to record information which will then be displayed when the [APD file](#) is formatted and examined.

```
import com.ocsystems.aprobe.*;

public class Example3 extends ProbeMethod
{
    public boolean onEntry( Object[] parameters )
    {
        Logger.log( "Hello world (from my probe)!" );
        return true;
    }

    public Object onExit( Object returnValue )
    {
        Logger.log( "Goodbye world (from my probe)!" );
        return returnValue;
    }

    public void onExit() // no return value
    {
        Logger.log( "Goodbye world (from my probe)!" );
    }

    public void onExceptionExit( Throwable e )
    {
        Logger.log(
            "Goodbye world via exception (from my probe)!" );
    }
}
```

Example3.java

The onLine() Method

The probes that we have seen so far perform actions only on entry to, and exit from, the target methods. It is also possible to probe individual source lines of a method *if it has been compiled with debugging information* (usually using “javac -g”). This is done using the method `onLine()`, as shown in [Example4.java](#) below.

```
import com.ocsystems.aprobe.*;

public class Example4 extends ProbeMethod
{
    private String methodName = "(unknown)";

    private void printEvent( String event )
    {
        System.out.println(
            event + " (in method " + methodName + ")" );
    }

    public void onLine( int lineNumber )
    {
        printEvent( "Line # " + lineNumber );
    }

    public boolean onEntry( Object[] parameters )
    {
        methodName =
            SymbolTable.getPrintableMethodName( getMethodId() );
        printEvent( "Method Entry" );
        return true;
    }

    public Object onExit( Object returnValue )
    {
        printEvent( "Method Exit" );
        return returnValue;
    }

    public void onExit()
    {
        printEvent( "Method Exit" );
    }

    public void onExceptionExit( Throwable e )
    {
        printEvent( "Method Exit via exception" );
    }
}
```

```
}
```

Example4.java

Enabling Line Probes

If not applied selectively, line probes can seriously impact performance. For this reason, line probing is turned *off* by default.

Thus, in order for the `onLine()` method to be called, you must specify that your probe needs to access lines. This is done in the deployment descriptor (`.xmj`) file, using the attribute “lines” in the `<probe>` tag, as shown in [Example4.xmj](#) below. Valid values for the “lines” attribute are “TRUE” or “FALSE”. If you do not specify a value, “FALSE” is assumed.

```
<probe_deployment>
  <probe class="Example4" lines="TRUE">
    <target value="Pi::main" />
  </probe>
</probe_deployment>
```

Example4.xmj

Advanced Custom Java

So far, all the examples have been, in certain ways, pretty simple and straightforward. Each probe is independent. Each applies to certain methods, and a new probe instance is created every time one of those methods is invoked. You've simply specified the name of the probe, and its targets, in the deployment descriptor, and RootCause did the rest automatically.

Under the covers, however, what's going on is quite intricate. For each method being probed, there exists what we call a "trigger", which, essentially, works as a "factory" for probes. When the method is invoked, this trigger gets, well, triggered, and then, by default, creates an instance of the appropriate probe.

This structure allows enormously powerful customizations. Triggers can create probes only conditionally. Triggers can be enabled or disabled. Triggers can be removed entirely from association with a method, and new triggers can be created dynamically.

ProbeBeans

How does one use this power? With a "ProbeBean". What is a "ProbeBean"? It is a class, derived from `com.ocsystems.aprobe.ProbeBean`, that groups together related code, data, triggers, and probes. ProbeBeans are created and initialized when the Java application first starts (while RootCause is on), and they in turn create the triggers that create the probes. Every probe is created by a trigger, and every trigger belongs to a particular ProbeBean.

There are a couple of places where you would want to use a "ProbeBean" instead of the simpler method described earlier:

- you can collect data on a per-thread basis; and
- sophisticated probes can be structured more easily, e.g., the number of class files can be reduced for the probes and data can be more easily shared among probes.

In the examples we've seen so far, an automatically-created ProbeBean created triggers to invoke the probes you specified in your deployment descriptor file. The files [MyFirstProbeBean.java](#) and [MyFirstProbeBean.xml](#) below show how you would manually create and deploy a ProbeBean for [MyFirstProbe.java](#), shown at the start of this chapter.

```
<probe_deployment>
  <bean class="MyFirstProbeBean">
    <instrument_target>
      <target value="Pi::main"/>
    </instrument_target>
  </bean>
</probe_deployment>
```

MyFirstProbeBean.xmlj

In [MyFirstProbeBean.xmlj](#) we use the tag `<bean>`, and the attribute “class” to specify which class is the ProbeBean to load. The `<target>` tag looks familiar, but what is `<instrument_target>`? Since the deployment descriptor no longer directly causes the creation of probes, we don’t know which classes we need to add “hooks” to. The `<instrument_target>` tag indicates that RootCause must add its hooks to the targets (or target lists) specified.

Note that `<instrument_target>` does *not* create any triggers or probes, the way a `<probe>` tag does. The creation of triggers and probes is left up to the ProbeBean itself, as shown in [MyFirstProbeBean.java](#) below.

The tag `<instrument_target>` also accepts the attribute “lines” just as “probe” does, so that line probes may be added.

```
import com.ocsystems.aprobe.*;
public class MyFirstProbeBean extends ProbeBean
{
    public void onEntry()
    {
        int targetID = getTarget();

        new ProbeTargetTrigger( targetID )
        {
            public Probe createProbe()
            {
                return new MyFirstProbe();
            }
        };
    }
}
```

```
}
```

MyFirstProbeBean.java

Let's look more closely at `MyFirstProbeBean`, which is as simple as it gets. When the deployment descriptor is read, `RootCause` will create an instance of any `ProbeBean` specified with the `<bean>` tag, and invoke its `onEntry()` method. The `ProbeBean` needs to know what methods to apply itself to, so it can create the appropriate triggers. This is done with the method `getTarget()`, which returns an ID corresponding the group of methods specified within an `<instrument_target>` tag in the `.xmj` file. The `ProbeBean` then creates a new trigger, derived from `ProbeTargetTrigger` and applied to that target, which in turn creates probes of the class `MyFirstProbe`. The fact that our new trigger is a `ProbeTargetTrigger` causes it to automatically apply itself to all the methods represented by the target ID, and its `createProbe()` method gets invoked whenever any of those methods does.

That's probably more than you wanted to know, but this allows you to explore the power of `ProbeBeans` further in the next example.

Parameterizing Probes

Once you've compiled your `ProbeBean` and `Probe` Java code, you can still greatly vary their behavior through the deployment descriptor. The obvious way we've already seen is by changing the target methods to which your probes apply. You can also, however, pass arbitrary other parameters to your `Probe` or `ProbeBean` via the deployment descriptor, using the `<parameter>` tag. This pair of examples will show how to do that, for a `ProbeBean` or for a stand-alone `Probe`.

```
<probe_deployment>
  <bean class="Example6ABean">
    <parameter name="ReallyHaveProbes" value="true"/>
    <instrument_target>
      <target value="Pi::main"/>
    </instrument_target>
  </bean>
</probe_deployment>
```

Example6a.xmj

As you can see, the `<parameter>` tag has two attributes, “name” and “value”; both are required. In [Example6ABean.java](#) below, we’ll see how the `ProbeBean` can reference and use the parameter we’ve defined.

```
import com.ocsystems.aprobe.*;

public class Example6ABean extends ProbeBean
{
    public void onEntry()
    {
        String reallyHaveProbesString =
            getParameter( "ReallyHaveProbes" );
        boolean reallyHaveProbes =
            ( Boolean.valueOf( reallyHaveProbesString )
              ).booleanValue();

        if( reallyHaveProbes )
        {
            int target = getTarget();

            new ProbeTargetTrigger( target )
            {
                public Probe createProbe()
                {
                    return new Example4();
                }
            };
        }
        else
        {
            System.out.println(
                "Probes disabled by parameter in deployment!" );
        }
    }
}
```

Example6ABean.java

The ProbeBean retrieves the value of a parameter, as a String, by name, also a String. This name passed to `getParameter()` must exactly match the attribute “name” in the `<parameter>` tag.

This method of parameterization via the deployment file may also be used without ProbeBeans, as shown on the following pages:

```
<probe_deployment>
  <probe class="Example6BProbe">
    <parameter name="BeVerbose" value="true"/>
    <target value="Pi::main"/>
  </probe>
</probe_deployment>
```

Example6b.xml

The parameter is specified the same way here, but nested within the “probe” tag rather than the <bean> tag. The mechanism by which a Probe retrieves the parameter is shown in [Example6BProbe.java](#).

Every probe has a field which references the trigger that created it, and every trigger belongs to a bean. Here, even though we did not create a custom bean explicitly, a default bean was created for us by RootCause. It is through this bean that a probe accesses its parameters. Here, the probe uses the value of the parameter to decide how much information to display.

Note that although both of these examples used the parameter as a boolean value, you are not restricted to that. Parameters could represent names, numerical values, etc. You may have multiple parameters for a single bean or probe, as long as each parameter has a unique name.

```
import com.ocsystems.aprobe.*;

public class Example6BProbe extends ProbeMethod
{
    boolean beVerbose;

    public boolean onEntry( Object[] parameters )
    {
        System.out.println( "Entering a method." );

        String beVerboseString =
            trigger.bean.getParameter( "BeVerbose" );
        beVerbose =
            ( Boolean.valueOf( beVerboseString )
              ).booleanValue();

        if( beVerbose )
        {
            for( int i=parameters.length-1; i>=0; i-- )
            {
                System.out.println(
                    "Parameter # " + i + " is " + parameters[i] );
            }
        }

        return true;
    }

    public Object onExit( Object returnValue )
    {
        System.out.println( "Exiting a method." );

        if( beVerbose )
        {
            System.out.println(
                "Method's return value is: " + returnValue );
        }

        return returnValue;
    }

    // exercise for reader:
    // provide other onExit, onExceptionExit methods
}
```

Example6BProbe.java

Working with Threads

In addition to methods, RootCause can also track threads. In fact, unless you specify otherwise, all your method probes are actually created within the context of a default thread probe! But you can also create your own thread probes to get more information about your multi-threaded applications. Thread probes are created by a thread trigger every time the JVM creates a new thread. The thread trigger is typically created via a ProbeBean.

You can use thread probes in order to keep track of data on a per-thread basis. Like method probes, thread probes have an `onEntry()` method. All thread probes derive from the class `com.ocsystems.aprobe.ProbeThread`.

A thread probe is created by a `ProbeThreadTrigger`. Our `ProbeBean` creates a new such `ProbeThreadTrigger` that will create instances of our `ProbeThread` class.

[Example7Bean.java](#), on the next page, tracks the beginning of all threads, and counts calls to methods within each of those threads. The [Example7.xmlj](#) deployment description file applies this thread tracking to all methods in the `Pi` class.

```
<probe_deployment>
  <bean class="Example7Bean">
    <instrument_target>
      <target value="Pi::*" />
    </instrument_target>
  </bean>
</probe_deployment>
```

Example7.xmlj

```
import com.ocsystems.aprobe.*;

public class Example7Bean extends ProbeBean
{
    int target;
    static int numberOfThreads = 0;

    class Example7ThreadProbe extends ProbeThread
    {
        int threadNumber;
        int callsInThisThread;

        class Example7MethodProbe extends ProbeMethod
        {
            {
                public boolean onEntry( Object[] p )
                {
                    callsInThisThread++;
                    System.out.println(
                        "Call # " + callsInThisThread +
                        " in thread # " + threadNumber );
                    return true;
                }
            }
        }

        public void onEntry() // thread entry
        {
            threadNumber = ++numberOfThreads;
            System.out.println(
                "Hello from start of thread # " + threadNumber );

            new ProbeTargetTrigger( target, this )
            {
                public Probe createProbe()
                {
                    return new Example7MethodProbe();
                }
            };
        }
    } // end ProbeThread
}
```

```
public void onEntry() // bean entry
{
    target = getTarget();

    new ProbeThreadTrigger()
    {
        public Probe createProbe()
        {
            return new Example7ThreadProbe();
        }
    };
}
```

Example7Bean.java**Dynamic Probe
Deactivation**

Triggers include the ability to be activated and deactivated dynamically.

[Example8.java](#) on the next page shows a probe that will only be called once per thread, because the first time it is invoked, it disables the trigger that created it.

Of course, you don't have to restrict your logic to turning the trigger off immediately. You could set whatever conditions you like, such as after 100 iterations, or when a buffer fills up.

```
import com.ocsystems.aprobe.*;

class MyMethodProbe extends ProbeMethod
{
    public boolean onEntry( Object[] p )
    {
        System.out.println(
            "You should only see this once per thread!" );
        getTrigger().disableProbe();
        return true;
    }
}

public class Example8 extends ProbeBean
{
    public void onEntry() // bean entry
    {
        int target = getTarget();
        new ProbeTargetTrigger( target )
        {
            public Probe createProbe()
            {
                return new MyMethodProbe();
            }
        };
    }
}
```

Example8.java

Index

A

actions, 3-9, 8-24

add dynamic module, 8-5

Add From Data Files To Display, 8-36

Add From Index To Display, 8-36

Add Process, 8-30

Add Process Data, 8-34

Add Process Data Dialog, 8-30

Add Selected Process Data, 8-36, 10-9

Add UAL, 8-7

Add Ual Dialog, 8-11

Additional Aprobe Options, 8-15

ADI file, 3-9

agent, 3-9

 RootCause, 2-1, 2-7

agent license, 6-3

agent_license.dat, 6-3

AIX, 2-2, 4-4, 5-10, 7-8, 8-47

apaudit, 4-4

APC, 3-9

APD file, 3-3

 name, 8-14

 number, 8-14

 size, 8-14

APD ring, 3-3, 3-9, 8-14

apformat, 3-3

Apformat Options Tab, 8-15

Apformat Parameters, 8-11, 8-18

APP_START, 5-4, 7-3, 8-36, 8-40

APP_TRACED, 7-3, 8-36, 8-40

application, 3-2, 3-9

Application, in Trace Index, 8-32

Apply button, 8-26

APROBE, 7-5

aprobe, 7-8

 -d option, 8-14

 -k option, 8-14

 -n option, 8-14

 -qstack_size option, 8-15

 -s option, 8-14

 secure, 10-6

 -t option, 8-14

Aprobe Options Tab, 8-14

Aprobe Parameters, 8-11, 8-17

aprobe.jar, 11-3

APROBE_HOME, 2-8, 7-4, 7-5

APROBE_JAVA_HEAPSIZE, 7-5

APROBE_JRE, 7-5, 8-47

APROBE_LOG, 2-8, 4-2, 7-6

APROBE_MONOSPACED_FONT, 8-48

APROBE_REGISTRY, 2-8, 4-2, 7-3

APROBE_SEARCH_PATH, 7-6
APROBE_WM_WORKAROUND, 8-48
Autoload Output, 8-19
.aws directory, 3-13

B

background color, 8-49
bash, 9-2
blue dots, 8-22
Build, 5-14
Build Options Tab, 8-14
Build, after Save As, 8-4
Build, in Setup menu, 8-7
build, rootcause subcommand, 9-4
buttons, arrow, 8-39

C

Call Stack Pane, 8-42
call tree, 5-13
CALL_FROM, 8-40
capacity options, 8-15
CD-ROM, 2-5
Change, 8-34, 8-35
.class file, 8-29, 11-3
Class Path, 8-22
 setting, 8-16
.clct file, 3-10, 6-4, 7-2, 9-5
<clinit> method, 8-22
clipboard, 8-6, 8-49
collect
 definition, 3-9
 file, 7-2
 rootcause subcommand, 9-5
Collect File, 8-21
color, background, 8-49
com, in Java package, 8-22, 8-23
com.ocsystems.aprobe Java package, 11-3
com.ocsystems.aprobe.Logger, 11-7
com.ocsystems.aprobe.ProbeMethod, 11-6
com.ocsystems.aprobe.ProbeThread, 11-17
com.ocsystems.aprobe.SymbolTable, 11-5
COMMENT, 8-25, 8-41
comment, logging, 8-25
compatibility, 2-6
Consider Case, 8-27, 8-35
constructor, 8-22
Copy UAL, 8-11

copy/paste, 8-49
csh, 9-2
custom Java probes, 11-1
cut/paste, 8-6

D

Data File, 7-2
Data File Size, 8-12
Data Files Pane, 8-42
Data node in Workspace Tree, 8-3
data, preserving, 3-6
.dclct directory, 3-10, 6-4
decollect, 3-10, 9-16
Decollect Dialog, 8-21
decollection, 3-10, 7-2, 9-5
Delete Dynamic Module, 8-6
deploy, 1-2, 2-7, 3-2, 3-10, 9-8
Deploy Dialog, 6-2, 8-19
deployed workspace, 8-14
deployment descriptor file, 3-10, 7-2
Dereference Pointers, 8-27
Deselect Function In Trace Setup, 8-37
Disable Load Shedding for This Item, 8-23
disable tracing, 3-7, 8-25
Display N data files after selected, 8-14
Display N data files before selected, 8-13
Don't Trace All In, 8-23
Don't Trace This Item, 8-22
Don't Trace Wildcards, 8-28
Don't Load Shed Selected Functions, 8-45
Don't Prompt for Source Files, 8-15
Don't Trace All Lines In Function, 8-23
Don't Trace Selected Functions, 8-45
.dply file, 3-10, 6-2, 6-3, 9-18
DTD, for XMJ file, 7-2
dynamic module, 8-2, 8-5
dynamically loaded library, 3-10, 8-5

E

Edit Class Path Dialog, 8-16, 8-22
Edit Source Path Dialog, 8-16
Edit UAL, 8-7
Edit Wildcard Strings Dialog, 8-28
Enable Load Shedding for This Item, 8-23
Enable Tracing, 8-25
END_DISPLAYED_DATA, 8-40
ENTER, 5-13, 8-40

environment variables, 7-1

- AP_ROOTCAUSE_ENABLED, 7-8
- APROBE, 7-5
- APROBE_HOME, 7-5
- APROBE_JRE, 8-47
- APROBE_LOG, 2-8, 7-5
- APROBE_MONOSPACED_FONT, 8-48
- APROBE_REGISTRY, 2-8
- APROBE_SEARCH_PATH, 7-6
- APROBE_WM_WORKAROUND, 8-48
- LD_AUDIT, 7-6, 7-7
- RC_SHORT_WORKSPACE_LOC, 7-8
- RC_WORKSPACE_LOC, 7-8, 8-11

errors, ld.so.1, 10-3**event**

- Exceptions, 8-34
- finding in Trace Index, 8-33
- kind, 8-40
- selecting, 8-33
- Snapshots, 8-34
- stepping in, 8-38
- tree, 5-13

Event Details Pane, 8-42**Event Trace Tree, 3-10, 8-40****events, 8-31****Examine Process Data, 8-8****examples, 11-1****Exceed, X emulator, 8-47****EXCEPTION, 8-41****Exception event, 8-18****Exception Snapshots, 8-18****exceptions**

- Java, 5-8, 8-3
- logging, 8-18
- snapshot, 8-18
- UAL, 8-2, 8-19

EXCP event kind, 8-33**executable, 3-10****EXIT, 5-13, 8-40****F****FILE event kind, 8-33****Find, 8-35****Find All, 8-35****Find Button, 8-40****Find Class In Trace Setup, 8-37****Find Function In Trace Events, 8-38****Find Function In Trace Setup, 8-37****Find Function/Method, 8-23****Find In Index, 8-33****Find In Program Contents Dialog, 8-26****Find Source, 8-37****Find Source File, 8-24****Find Text In Events Dialog, 8-35****Find Text in Trace Events, 8-38****Find Text in Trace Events Dialog, 8-43****FLEXlm, 2-9****flight recorder, 6-2****format, 3-10****ftp, 6-3, 6-4****G****gcc, 2-4****gethrtime, 8-25****gethrvtime, 8-25****getMethodId(), 11-6****getPrintableMethodName(), 11-5, 11-6****Global Trace Options Dialog, 8-27****Glossary, 3-9****Goto File, 8-27****H****heap, Java, 7-5, 7-6****I****Index Process Data, 8-8****<init> method, 8-22****install_rootcause, 9-2****J****J2EE, 8-6, 8-10, 10-14****JAR, 8-22****.jar file, 8-29****JAR file in workspace, 11-3****Java, 2-4****\$java\$, 8-22****Java Exceptions Configuration Dialog, 8-18****Java Path Dialog, 8-17****Java probes, 7-2****Java version, 8-47****java version, 5-2****JAVA_CLASS_LOAD, 8-41**

JAVA_CLASS_LOADS, 8-41
java_exceptions UAL, 5-8, 8-3
JAVA_LOAD_SHED, 4-3, 8-41
JNI, 8-5
JRE, 3-11, 5-2, 8-17, 8-47, 8-49
JVM, 8-15

K

Keep logged data for N previous processes, 8-12
Kind, in Trace Index, 8-33
Korn shell, 2-3
ksh, 2-3, 9-2

L

Last Data Recorded, 8-31, 8-33
ld.so.1 error message, 10-3
LD_AUDIT, 10-5, 10-7
LD_AUDIT environment variable, 7-6, 7-7
LD_PRELOAD, 10-5
libaudit.so, 10-3
library, dynamically loaded, 3-10, 8-5
license, 2-8, 6-3
LINE, 8-41
Line Number, 8-27
Linux, 2-3, 2-5, 7-7
load shedding, 3-7
LOAD_SHED Table, 8-44, 8-45
local, 2-1, 2-7, 3-11
local disk, 3-2
log, 3-2, 3-11
 from Java, 11-7
Log Comment, 8-25
Log Java Class Loads, 8-27
Log Parameters checkbox, 5-16
Log Snapshot, 5-17, 8-25, 8-34
Log Statistic, 8-25
Log Traceback, 8-25
log(), 11-7
log_env, 8-2, 8-42
Logging Exceptions, 8-18

M

main class name, 8-15
Maximum Logged String Length, 8-28
Maximum number of events in Trace Display, 8-13

Maximum number of items in Trace Index, 8-13
method ID, 11-5
method names, 8-29
module, 3-11, 5-8
 dynamically loaded, 8-5
multi-program applications, 10-13
mwm, 8-47

N

Name of UAL, 8-11
names, for tracing, 8-29
New Class Dialog, 8-29, 8-37
New Workspace Dialog, 8-4, 8-9

O

onEntry, Java method, 11-2
onExceptionExit, Java method, 11-2
onExit, Java method, 11-2
onLine(), Java method, 11-8
Open Associated Workspace, 4-3, 8-36
open, rootcause subcommand, 9-15
Options button, 8-26
Options, in Setup menu, 8-7
overhead, 3-7

P

package, Java, 8-22, 8-23
parameters, logging, 8-24
PATH, 4-2
PDF, 2-2
performance, 3-2, 11-9
Pi class, 5-16
PID, 3-9, 3-11
Plug-in Class, 8-11
plug-in for UALs, 8-17
pointers, logging, 8-27
popup menu, 8-41
 Table Dialog, 8-43, 8-44, 8-45
 Trace Setup Dialog, 8-22
 Workspace Tree, 8-2
predefined UAL, 3-11, 8-2
 exceptions, 8-2
 java_exceptions, 8-3
 log_env, 8-2, 8-42
 sigsegv, 8-3

predefined UAL (continued)
 user-defined, 8-2
 verify, 6-4, 8-3
preferences file, 7-4
preserving data, 3-6
Probe Action, 8-25
probe deployment descriptor, 8-26
probe names, 8-29
Probe Trigger, 8-24
ProbeBean, 11-10
ProbeMethod, 11-5
probes, 3-2, 3-11
Probes Pane, 5-17, 8-24
ProbeThreadTrigger, 11-17
PROCESS, 8-40
PROCESS DATA, 8-3
Process Data Set, 3-5, 8-3, 8-31
Process Statistics, 8-25
Process, in Trace Index, 8-32
program, 3-2, 3-12
 changed, 8-5
 node in Workspace Tree, 8-2
 run, 8-8
Program Contents Tree, 5-8, 8-21
Program Information Pane, 8-42
PROGRAM_COMMENT, 8-41

R
rclog, 7-5
red dot, 8-23
Reflection, X emulator, 8-47
Refresh, 8-36
Refresh Index, 8-33
register, 3-12, 5-6, 8-5
 rootcause subcommand, 9-17
Register Program, 8-5
registry, 3-12, 9-17
Release Notes, 1-3
remote, 2-1, 2-7, 3-12
 computer, 9-5
 workspace, 8-14
Remove Probes For All Child Items, 8-23
Remove UAL, 8-7
Requires Trace UAL, 8-11
Reset Dynamic Module, 8-5
Reset Program, 8-5
Reset Program Dialog, 8-10
Root Java Module, 8-22

RootCause
 Agent, 3-2, 9-5, 9-17, 9-18
 Console, 3-2
 starting, 9-15
RootCause agent, 2-1, 2-7
rootcause commands, 9-3
 build, 9-4, 11-4
 collect, 6-3, 7-2, 9-5
 config, 9-6
 decollect, 9-7
 deploy, 9-8
 log, 9-11
 merge, 9-12
 new, 9-13
 off, 9-14
 on, 6-3, 9-14
 open, 4-2, 9-15
 register, 7-3, 9-17
 run, 4-4, 9-19
 status, 9-19
 xrun, 9-19
RootCause Console, 2-1
rootcause format, 3-10
RootCause Log, 3-11, 4-2, 8-4, 9-5
 decollected, 6-4
 size, 9-11
RootCause Main Window, 5-7
RootCause Options Dialog, 8-12
RootCause Registry, 9-5
rootcause.properties, 8-49
rootcause_disable, 9-2
rootcause_enable, 9-2
rootcause_libpath, 10-4, 10-8
rootcause_off, 4-2, 7-6, 7-7, 7-8, 9-2
rootcause_on, 4-2, 5-10, 7-3, 7-6, 7-7, 7-8, 9-2
ROOTCAUSE_SNAPSHOT, 8-26
rootcause_status, 9-2
Run Options Tab, 8-15
Run Program Dialog, 8-19
run under RootCause, 3-12
run_with_apaudit, 4-4, 7-8
rusage, 8-25

S
saprobe, 10-6
Save As Text, 8-37, 10-9
Save As XML, 8-37, 10-9
Save As, workspace, 8-4

- Search All Modules, 8-27
- search path, 7-6
- secure aprobe, 10-6
- secure_applications file, 10-6
- Select Data Files, 8-33
- Select Data Files Dialog, 8-34
- Select Events, 8-33
- Select Events Dialog, 8-34
- Setup Menu, 8-6
- setup script, 4-2
- sh, 9-2
- shadow header file, 3-12
- shadow header files, 10-11
- shared library, 3-12, 8-5
- shell, Unix, 9-2
- sigsegv, predefined UAL, 8-3
- size of RootCause Log, 9-11
- SNAP event kind, 8-33
- snapshot, 3-4, 3-6, 8-18
 - probe, 8-25
- Solaris, 2-3, 2-5, 7-6, 8-47
- Solaris version, 8-6, 8-47, 8-49
- sort, table column, 8-32
- source files, finding, 7-6, 8-7, 8-16
- Source Options Tab, 8-15
- Source Pane
 - Trace Display, 8-42
 - Trace Setup, 8-24
- Source Path, setting, 8-16
- START_DISPLAYED_DATA, 8-40
- START_OF_TRACE, 8-41
- starting the GUI, 9-15
- statistics, logging, 8-25
- Step Into Backwards, 8-39
- Step Into Forward, 8-38
- Step Menu, 8-38
- Step Next Backward, 8-38
- Step Next Forward, 8-38
- Step Out Backwards, 8-39
- Step Out Forward, 8-39
- Step Over Backwards, 8-39
- Step Over Forward, 8-39
- Stepping Toolbar, 8-39
- stepping, in trace events, 8-38
- strings, logging, 8-28
- stripped, 3-12
- SYN_CALL_COUNTS, 8-41
- SYN_JAVA_CALL_COUNTS, 4-3, 5-13, 5-14, 8-41, 8-43

T

- table, 8-43
 - CALL_COUNTS, 8-41, 8-43
 - JAVA_CLASS_LOADS, 8-41, 8-44
 - LOAD_SHED, 8-44
 - Show Associated, 8-38
 - sorting, 8-32, 8-45
 - Trace Index, 8-31
- Table Dialog, 8-43
- target method ID, 11-5
- TEXT, 7-3, 8-41
- text, save as, 8-37
- Thread, in Trace Index, 8-33
- THREAD_END, 8-40
- THREAD_START, 8-40
- threads, Java, 11-17
- threads, view events by, 8-38, 8-40
- Time, in Trace Index, 8-32
- time, logging, 8-25
- time, view events by, 8-38, 8-40
- Total logged data limit per process, 8-13
- Trace All In, 8-22, 8-23
- Trace All Lines In Function, 8-23
- Trace Data Dialog, 8-30
- Trace Data Files, 8-30
- Trace Display, 8-35
- Trace Index Dialog, 3-10, 8-25, 8-31
- Trace Setup Dialog, 4-3, 5-8, 5-13, 8-21
 - colored dots in, 8-22
 - Popup Menu, 8-22
 - Probes Pane, 8-24
 - Program Contents Tree, 8-21
 - Source Pane, 8-24
 - Variables Pane, 8-24
- Trace This Item, 8-22
- trace UAL, 8-2
- Trace Wildcards, 8-28
- TRACEBACK, 8-41
- traceback, 8-3, 8-42
 - logging, 8-25
 - overhead of, 8-25
- tracing, 3-13
 - disabling, 3-7, 8-25
- tree
 - event trace, 8-40
 - program contents, 8-21
 - Workspace, 8-1
- trigger, 3-13, 8-24
- tuning a trace, 5-13

U

UAL, 3-13, 8-2
 node in Workspace Tree, 8-2
 predefined, 8-2
UAL Description, 8-11
UAL File, 8-11, 8-17
UAL Name, 8-17
UAL Options Dialog, 8-17
Unregister Program, 8-5
Update, 8-34, 8-35

V

Variables Pane, 8-24
verify UAL, 6-4, 8-3
version compatibility, 2-6
View Menu, 8-38

W

white papers, 1-3
wildcard
 in Trace Setup, 8-29
 in XMJ file, 11-4
Workspace
 Browser, 8-1
 Edit Menu, 8-6
 Execute Menu, 8-7
 Help Menu, 8-8
 Message Pane, 8-3
 Toolbar, 8-8
 Tree, 8-1
 Workspace Menu, 8-3
workspace, 3-2, 3-13, 5-5, 9-13
 JAR file in, 11-3
 location, 7-8
 rebuilding, 11-4
Workspace Tree, 8-1
 Data, 8-3
 Popup Menu, 8-2
 Program, 8-2
 UALs, 8-2
Wraparound data logging wraps at N, 8-13

X

X windows, 8-47
.xmj file, 3-10
XMJ file, 7-2, 11-3
XML, 3-13
 Save As, 8-37
XWindows emulators, 8-47