

WIND RIVER

# Wind River® Workbench Function Tracer

USER'S GUIDE

3.0

---

Copyright © 2007 Wind River Systems, Inc.

All rights reserved. No part of this publication may be reproduced or transmitted in any form or by any means without the prior written permission of Wind River Systems, Inc.

Wind River, the Wind River logo, Tornado, and VxWorks are registered trademarks of Wind River Systems, Inc. Any third-party trademarks referenced are the property of their respective owners. For further information regarding Wind River trademarks, please see:

<http://www.windriver.com/company/terms/trademark.html>

This product may include software licensed to Wind River by third parties. Relevant notices (if any) are provided in your product installation at the following location:  
*installDir\product\_name\3rd\_party\_licensor\_notice.pdf.*

---

#### **Corporate Headquarters**

Wind River Systems, Inc.  
500 Wind River Way  
Alameda, CA 94501-1153  
U.S.A.

toll free (U.S.): (800) 545-WIND  
telephone: (510) 748-4100  
facsimile: (510) 749-2010

For additional contact information, please visit the Wind River URL:

<http://www.windriver.com>

For information on how to contact Customer Support, please visit the following URL:

<http://www.windriver.com/support>

# Contents

<b>1</b>	<b>Introduction .....</b>	<b>1</b>
1.1	Introduction .....	1
1.2	Architectural Summary .....	2
1.3	Features .....	4
<b>2</b>	<b>Getting Started .....</b>	<b>7</b>
2.1	Introduction .....	7
2.2	Requirements .....	7
	Host .....	8
	Target .....	8
2.3	Starting Function Tracer .....	9
	Target Considerations .....	10
	Starting Function Tracer From Workbench .....	11
	Unresolved Symbols at Startup .....	13
2.4	Testing Your Installation .....	14
<b>3</b>	<b>The Function Tracer GUI .....</b>	<b>23</b>
3.1	Introduction .....	23

<b>3.2</b>	<b>The Function Tracer GUI .....</b>	<b>23</b>
3.2.1	Registration Window .....	24
	Window Elements .....	25
3.2.2	Main Window .....	28
	Window Elements .....	29
3.2.3	Source Code View Window .....	33
	Source Path Dialog Box .....	35
	File Search Dialog Box .....	36
3.2.4	Snapshot Window .....	37
3.2.5	Console Window .....	39
	Window Elements .....	39
3.2.6	Highlight Window .....	40
	Window Elements .....	41
	Configuring Highlight Criteria .....	42
	Example .....	43
3.2.7	Columns Dialog Box .....	44
3.2.8	Custom Modules Dialog Box .....	45
	Window Elements .....	46
	Loading Custom Modules .....	47
<b>4</b>	<b>Using Function Tracer .....</b>	<b>49</b>
4.1	Introduction .....	49
4.2	Starting Tracing Activity .....	49
	Initializing Trace Points .....	50
	Registering Trace Points .....	51
	Activating Trace Points .....	54
	Deactivating .....	55
	Modifying .....	55
	Removing .....	56
4.3	Viewing Data .....	56
	Overview Table .....	56

	Trace Table .....	58
	Detail Table .....	61
	Buttons .....	64
	Status Bar .....	65
<b>4.4</b>	<b>Operational Features .....</b>	<b>66</b>
	Viewing Source Code .....	66
	Arranging Columns .....	66
	Setting Highlight Color .....	67
	Taking Snapshots .....	67
	Adding Custom Modules .....	67
	Viewing Messages in the Console Window .....	67
<b>4.5</b>	<b>System Viewer Event Integration .....</b>	<b>68</b>
	Automatic System Viewer Support .....	69
	High Resolution Time-Stamp Driver .....	69
	System Viewer Events .....	69
<b>5</b>	<b>Usage Tips .....</b>	<b>71</b>
<b>5.1</b>	<b>Introduction .....</b>	<b>71</b>
<b>5.2</b>	<b>Observing Practical Limitations .....</b>	<b>72</b>
	Missing Symbols .....	72
	Processor Load and Bandwidth Considerations .....	72
	Routines That Must Not Be Traced .....	73
	Tracing Frequently Called Routines .....	75
<b>5.3</b>	<b>Tracing Tips .....</b>	<b>76</b>
	Tracing Routines Returning Floating Point Values .....	76
	Tracing Real-Time Processes .....	76
<b>6</b>	<b>Troubleshooting .....</b>	<b>77</b>
<b>6.1</b>	<b>Introduction .....</b>	<b>77</b>
<b>6.2</b>	<b>Loading and Initializing Function Tracer Manually .....</b>	<b>78</b>
	Load the Required Libraries .....	78
	Initialize Function Tracer .....	79
	Example Target Script .....	80

	Starting Function Tracer Manually from the Command Line .....	81
<b>6.3</b>	<b>General Troubleshooting Tips .....</b>	<b>81</b>
	Issues With the Target .....	82
	Issues With the Host .....	85
<b>A</b>	<b>API Reference .....</b>	<b>91</b>
<b>B</b>	<b>Glossary .....</b>	<b>93</b>
	<b>Index .....</b>	<b>97</b>

# 1

## *Introduction*

1.1	Introduction	1
1.2	Architectural Summary	2
1.3	Features	4

### 1.1 Introduction

Real-time systems tend to be event-driven, multi-threaded applications that need to respond to external conditions. It is difficult to determine execution code paths especially when they jump between different threads. Simply setting breakpoints and examining call stacks does not provide sufficient thread-interaction information, and stopping the code on breakpoints is detrimental to real-time systems.

Wind River Function Tracer is a dynamic execution-tracing tool for use in developing embedded software. It monitors all calls to the functions you have registered, providing a live summary of each call as it occurs, which helps you pinpoint problems by showing the call sequences that lead to specific function calls. Function Tracer makes it easy to visualize how your programs execute.

#### Overview

Function Tracer helps you analyze and debug your system without special compilation. It provides run-time information, such as the following:

- Whether a traced function is ever called.
- When a traced function is called, relative to all other traced calls.
- When a traced function exits, relative to all other traced calls.
- Which task made each call.

For each traced function, Function Tracer records the following items:

- the task that called the function
- the parameters with which the function was called (optionally)
- the call stack that led up to the function call
- the return value from the function (optionally)
- the execution time for the function (optionally)

Armed with the above information, you can refine tracing-on-the-fly by adjusting filters, activating and deactivating trace points, and so forth, to pinpoint problem areas quickly and efficiently.

Because Function Tracer can trace functions without special compilation or linking, it makes all these capabilities available also for third-party code and operating system functions.

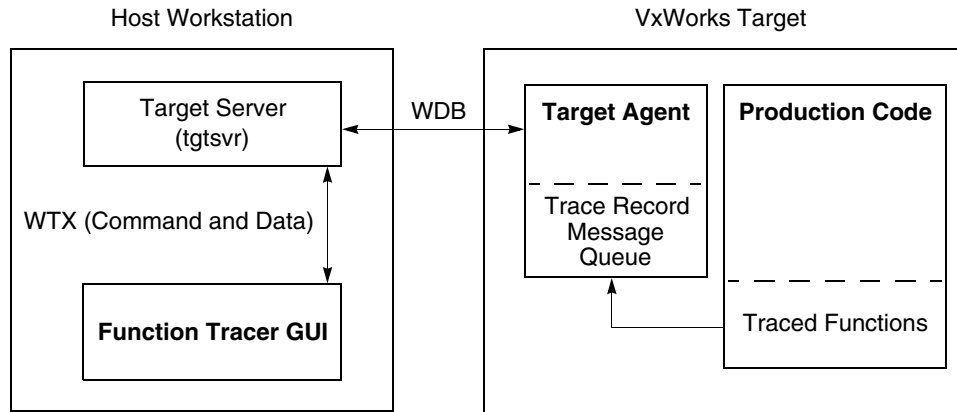
Function Tracer requires no special compilation of your code. It patches code with active trace points that record call parameters, return values, task information, and the call stack every time an active trace point is called. Function Tracer transmits the trace records to the host, where the graphical interface displays them for immediate viewing. When a trace point is deactivated, Function Tracer unpatches the function and slips out seamlessly.

## **1.2 Architectural Summary**

Function Tracer consists of two main subsystems:

- host-based graphical display and control tool (Function Tracer GUI)
- target-based patching and data collection





You interact using the host tool to manage trace points. The host tool sends only the activation and deactivation commands to the target agent.

When the target agent receives an activation command, it patches the specified function with code that creates a trace record upon entry to the specified function and another record upon exit. Each trace record (or **trace point**) can be configured to contain the information you need to understand how the function was called. For details on how to set up trace points, see [Registering Trace Points](#), p.51.

The following data items are sent to the host in a trace point for analysis and display.

- unique sequence ID that pairs function-entry with function-exit records
- task ID of the calling task
- optionally, the value of all the function parameters (upon entry), subject to the maximum number of parameters set during initialization
- optionally, the return value of the function call (upon exit)
- full call stack active at the time of the call (upon entry)
- execution time of the function, in milliseconds, if a high resolution timestamp driver exists on the target

When the target agent receives a deactivate command, it unpatches the specified function, returning the function to its original state

The target agent posts each trace record to a message queue on the target

The GUI running on the host computer periodically retrieves messages from the trace-record message queue on the target; it uses the WTX protocol using the target server. Function Tracer then processes these records on the host and displays the analyzed data in the GUI.

Function Tracer requires no special compilation of your code. It patches code with active trace points that record call parameters, return values, task information, and the call stack every time an active trace point is called. Function Tracer transmits the trace records to the host, where the graphical interface displays them for immediate viewing. When a trace point is deactivated, Function Tracer unpatches the function and slips out seamlessly.

## 1.3 Features

Function Tracer features include:

### **Live Graphical Trace**

Function Tracer dynamically shows you the traced functions as your system runs.

### **Ease of Use**

Function Tracer requires no special recompilation, relinking, source-code instrumentation, or special hardware. It can analyze already running code. Its intuitive graphical user interface gives you direct access to all the options and capabilities with a few clicks.

### **The Whole Picture**

Because it requires no special compilation, Function Tracer can analyze code you did not write, including operating-system code and third-party libraries.

### **Minimal Impact**

Function Tracer, being a software tool, has some impact on the execution of your programs. While designed to be as small as possible, the delay can become excessive if a traced function is executed repeatedly at a high frequency and/or if a large number of functions are traced. Trace records are queued and sent to the host as a background task, minimizing the impact on your code. Function Tracer operates using the same compiled code as your production system.

### **Dynamic Activation**

You can activate, deactivate, and change trace points dynamically (as your system runs) and view the results immediately. When you deactivate a trace point, the corresponding function is restored to its original state.

### **Snapshot Comparison**

With the GUI, you can take a trace-log snapshot at any time. You can compare future activity against the snapshot.

### **Custom Filtering**

Custom filtering lets you limit logging to only areas of interest by allowing you to specify a list of ignored tasks or watched tasks for each trace point.



# 2

## Getting Started

- 2.1 Introduction 7
- 2.2 Requirements 7
- 2.3 Starting Function Tracer 9
- 2.4 Testing Your Installation 14

### 2.1 Introduction

This chapter takes you through the process of installing, setting up, and running Wind River Function Tracer on either a VxWorks or Linux target. It gives you enough information to begin using Function Tracer. At each step, references are made to the location in this manual of more detailed descriptions. For more information on using Workbench, see the *Wind River Workbench User's Guide*.

### 2.2 Requirements

You must connect to the target manager for your target in order to use Function Tracer. For documentation on the target manager, consult the *Wind River*

*Workbench User's Guide, VxWorks Version: Target Manager View*, as well as your platform User's Guide.

There are some dependencies Function Tracer places on your host operating system for resources that are specific to the target platform, summarized in the sections below.

## Host

Function Tracer requires some resources from your host operating system. The following is a list of those requirements:

- If you want to collect data from a real-time process (RTP), the RTP components in your kernel must include shared data region support. This support must be built into your kernel before running it on the target. To accomplish this, include the **INCLUDE\_SHARED\_DATA** component
- If you want to time the execution of routines you are tracing, you must include the timestamp support component **INCLUDE\_TIMESTAMP** found in the **hardware/peripherals/clocks** directory in the **Kernel Configuration** component tree window. Note that not all targets have timestamp timer support.
- The Function Tracer graphical user interface (GUI) is implemented in Java. As such, the minimum recommended amount of memory on all types of host machines is 256MB. This requirement may increase if you run other Java applications on the same machine.



---

**NOTE:** In Windows, if you have installed and subsequently uninstalled any version of Java JDK or JRE on your machine, you must ensure that they have been uninstalled correctly such that your system registry does not refer to non-existent Java directories.

---

## Target

Your target requires some resources from your host operating system. The following is a list of those requirements:

- You must run the Target Manager for your target board in order to use Function Tracer. For information on the target manager, see the *Wind River Workbench User's Guide, VxWorks Version: Target Manager View*, as well as your platform User's Guide.

- The use of the WDB agent is necessary. The easiest way to ensure that your VxWorks Image Project (VIP) has WDB support is to make sure one of the following kernel configuration Profiles is used in your project:
  - **PROFILE\_COMPATIBLE**
  - **PROFILE\_DEVELOPMENT**
  - **PROFILE\_ENHANCED\_NET**

For more information, see the *Wind River Workbench User's Guide, VxWorks Version: VxWorks Image Projects*.

- If your target board is running a PowerPC processor, you should build your VxWorks Image Project with the extended vector addressing option enabled. This option is in the **Components** tab, in the project **Kernel Configuration** view, under **operating system components > kernel components** in the tree. Right-click **Allow 32-bit branches to handlers** in the tree, then click **Include** to enable this option in your build. Targets with large memories typically load Function Tracer at an address beyond the limited 26-bit PC-relative addressing normally used.
- If your target board is running an x86 processor, the VxWorks Image Project, and all applications running on it, must be built with no compiler options which disable the generation of frame pointers.



---

**CAUTION:** This is the default for both Gnu and Diab (Wind River) compilers, but beware that if you change it, you will encounter problems.

---

- Wind River Run-Time Analysis Tools do not support connecting to a target using a **WDB\_TIPC** connection. This means that if you are working in an **AMP** environment, you can only connect the Run-Time Analysis Tools to core 0 in AMP mode.

For troubleshooting tips, see [6.3 General Troubleshooting Tips](#), p.81.

## 2.3 Starting Function Tracer

In most cases, Function Tracer can be started automatically from Workbench by following the procedures outlined in this section. If launching fails, however, you must determine which libraries to load yourself, load them, then initialize

Function Tracer manually. This procedure is described in detail in [6.2 Loading and Initializing Function Tracer Manually](#), p.78.

## Target Considerations



---

**NOTE:** Function Tracer cannot run on a simulator.

---




---

**CAUTION:** You *must* set the following options on your target before connecting Function Tracer.

---

## Enabling Symbols

To ensure that all required symbols are loaded on the target, follow these steps:

1. Right-click the target name and select **Properties** to open the **Target Connection** dialog box.
2. In the **Target Server Options** tab, click Edit in the **Advanced Target Server Options** group to open the **Advanced Target Server Options** dialog box.
3. In the **Symbols** tab view of this dialog box, click the **Load global and local symbols** button, then click **OK**.
4. Click **OK** to close the **Target Server** dialog box.
5. Connect to the target server using the Workbench icon (.

## Tracing Semaphores



---

**WARNING:** If you trace semaphore activity, you must use extreme caution in doing so. Semaphore calls occur sporadically at rates up to hundreds of times per second. Attempting to trace such high rate calls *can and will* cause catastrophic failure of the target operating system.

To minimize the chances of such failure when tracing semaphore activity, start Function Tracer with **Stack Depth** set to 2, and **Task Filtering** showing only 1 task of interest at any given time. Failure to use this configuration when tracing the semaphore API *will* cause catastrophic failure of the target operating system.

---

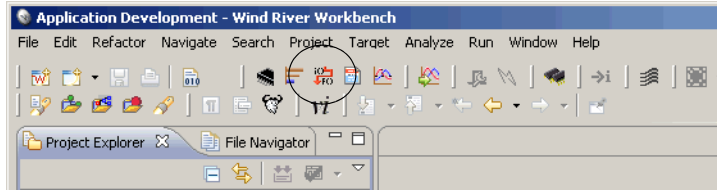
For more information on tracing semaphores, see [4. Using Function Tracer](#).



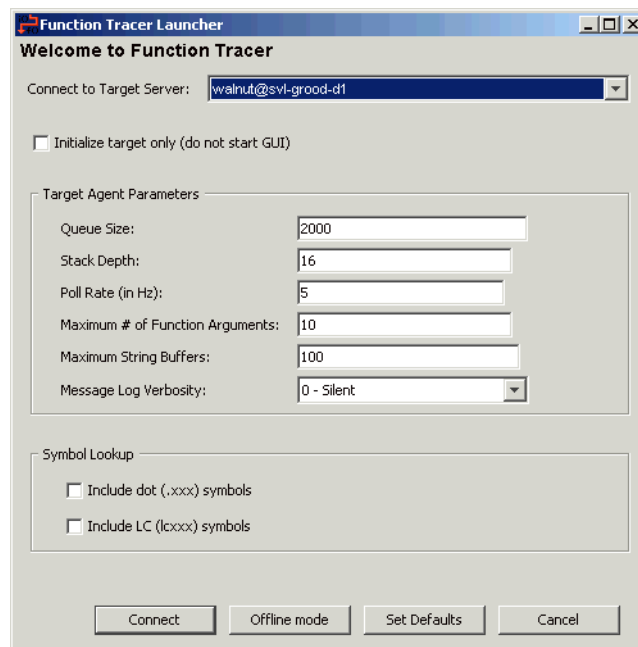
## Starting Function Tracer From Workbench

Follow these steps to load and start the Function Tracer GUI.

1. In Workbench, select the **Function Tracer** toolbar button (circled).



This opens the **Function Tracer Launcher** dialog box.



2. In the **Function Tracer Launcher** dialog box, enter values, or use the default values, for the following parameters to determine how to initialize the target.

### **Connect to Target Server**

Enter a live target server name (not a simulator) in the field, or choose one from the drop-down menu. The target server drop-down list is populated with those in the **Target Manager** view in Workbench. If a target is selected in this view, then it appears by default in the text entry field, but may be overridden with any other choice in the drop-down list.

### **Initialize target only (do not start GUI)**

Select this checkbox if you only start and initialize your target, without starting the Function Tracer GUI.

The **Target Agent Parameters** group contains the following parameters.

#### **Queue Size**

The target must queue a message for every trace point it encounters. These messages are queued on the target until they are retrieved by the host. If this queue is too small, messages will be lost, and trace records will be missed; in this event, the target agent notifies the host GUI. To avoid missing records, you may need to adjust this parameter, which should be entered in multiples of one thousand (1000, 2000, and so forth). The default is **2000** messages.

#### **Stack Depth**

Specify the depth of the call stack to be collected for each trace point. The call stack always starts from the lowest-level call, that is, the traced function. The default is **16**. Note that the larger this number is, the slower your system will run and the greater the bandwidth required when sending trace data to the host.

#### **Poll Rate (in Hz)**

Specify how often (in Hz) for the host to poll the target for trace records. During a single poll, the host continues reading until the target message queue is empty, then resumes polling at the specified rate. Polling too slowly causes excessive messages to build up on the target, but polling too rapidly consumes too much processing power on the host and target. The default is **5**.

#### **Maximum # of Function Arguments**

Specify the maximum number of input arguments to record for each trace point, in the range of 1-10. The default is **10**.

#### **Maximum String Buffers**

String arguments of traced functions must be copied and saved until retrieved by the host. Use this parameter to specify the maximum number of strings to be queued. The default is **100**.

### Message Log Verbosity

Specify the debug verbosity level for the GUI, in the range 0-3. Debug messages are sent to the **Console** window (see [3.2.5 Console Window](#), p.39), available from the **View** menu. A value of 0 causes only error messages to be logged. Using larger values (in the range of 1-3) causes increasingly more debug messages to be printed. The default is 0.



---

**CAUTION:** Setting target verbosity to a value greater than 0 may cause the target agent to needlessly generate a large number of messages.

**Generally, use the default value of 0 for target verbosity, unless requested by Wind River Technical Support to help you diagnose a problem.**

---

The **Symbol Lookup** group contains the following parameters.

#### Include dot (.xxx) symbols

In the attempt to resolve symbols, Function Tracer normally ignores symbols that begin with a dot ("."). To force these symbols to be resolved, check this box. The default is **unchecked**.

#### Include LC (lcxxx) symbols

(Same description as above, only for symbols starting with lc.)

3. Click **Connect**. This loads the libraries onto the target server automatically and initializes the target agent.
4. As an option to the above process, if you click **Offline mode**, Function Tracer does not connect to any target, but rather opens a **Console** window (see [3.2.5 Console Window](#), p.39) where you can navigate to, and open, a previously saved snapshot file instead. No target server parameters need to be entered if you choose this option.

When you launch the GUI, you must provide the name of a target server to which it should connect. Connection is successful only after the target libraries have been loaded and initialized. As a convenience, when the target reboots and target libraries have been loaded and initialized again, the GUI reconnects automatically and activates all trace points that were active at the time of reboot.

### Unresolved Symbols at Startup

When Function Tracer is loaded, it reports unresolved symbols if you do not have the high resolution timestamp driver, or **System Viewer** support, installed in the kernel. When you load Function Tracer automatically from Workbench (see [2.3 Starting Function Tracer](#), p.9), the **Function Tracer Setup** code detects whether

your target kernel includes the high resolution timestamp driver. If so, it initializes the Function Tracers target libraries with time-stamp support that enables Function Tracer to record precision timing information for each traced function.



---

**NOTE:** A few unresolved symbols present while Function Tracer is running is normal. This is caused by Function Tracer trying to link to optional services.

---

Once the GUI starts successfully, you can test your installation following the steps in the next section, or, if you wish, proceed to [3. The Function Tracer GUI](#) for instructions on using the GUI.

## 2.4 Testing Your Installation

You can trace the `malloc()` function to test whether or not Function Tracer is installed correctly.

To test your Function Tracer installation, do the following:

1. Start Workbench, then start the **Target Server**.
2. Open a host shell in Workbench to verify that the target server is connected to your target board.
3. Launch Function Tracer according to instructions in [2.3 Starting Function Tracer](#), p.9.

This opens the Function Tracer **Main** window and the **Registration** window to let you register trace points.



---

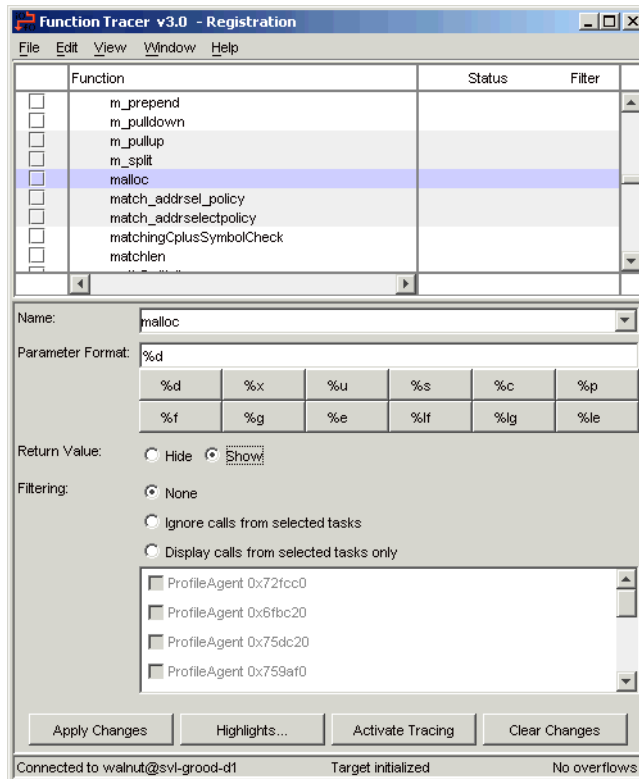
**NOTE:** If you have trouble starting Function Tracer in Workbench, you can follow the instructions in [6. Troubleshooting](#) for loading and running Function Tracer manually.

---

4. Verify that the status message at the bottom of the **Main** window reads: **Connected to target**.

If this message does not appear, open the **Console** window (see [3.2.5 Console Window](#), p.39) to check for error messages. You may want to restart Function Tracer with a verbosity setting of 1 or 2.

5. When validating VxWorks 6.6 text (executable code) addresses, Function Tracer automatically checks the text boundaries available for shared libraries, loaded modules, and the kernel. To facilitate locating all text ranges, ensure that all symbols are available to Workbench, and that module loader support is included in the target system project configuration.
6. In the **Registration** window, type **malloc** in the **Name** field.



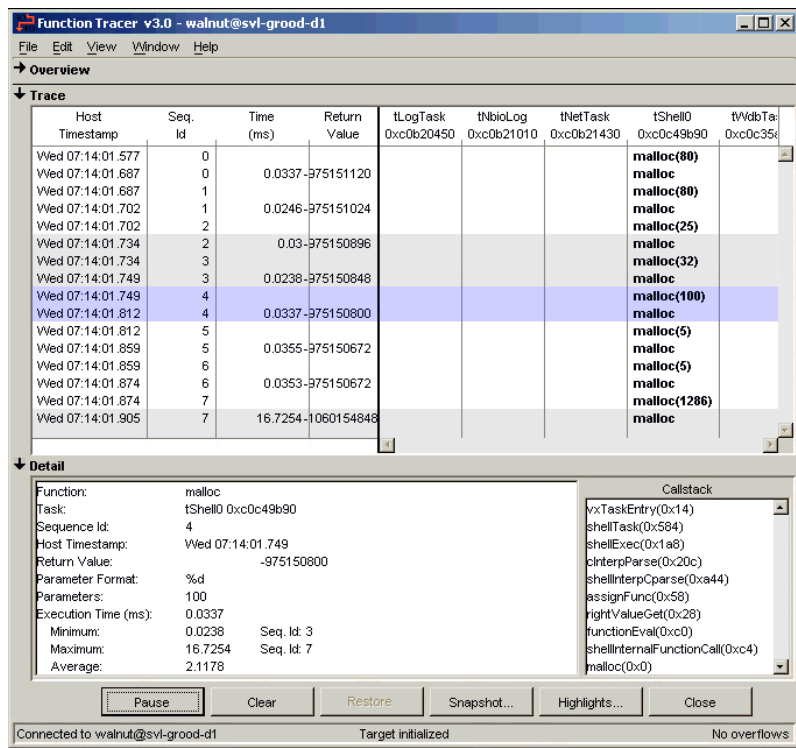
7. In the **Parameter Format** field, select the **%d** option (or type **%d**).
8. Select **Show** for **Return Value**, and select **None** for **Filtering**.
9. Click **Apply Changes**, then click **Activate Tracing** at the bottom of the **Registration** window. This registers and activates the trace point for **malloc()**.

10. If your target is running tasks that are calling `malloc()`, you should see trace records appearing in the Function Tracer **Main** window. If nothing is running, type in the command line window:

```
-> fish = malloc(100)
```

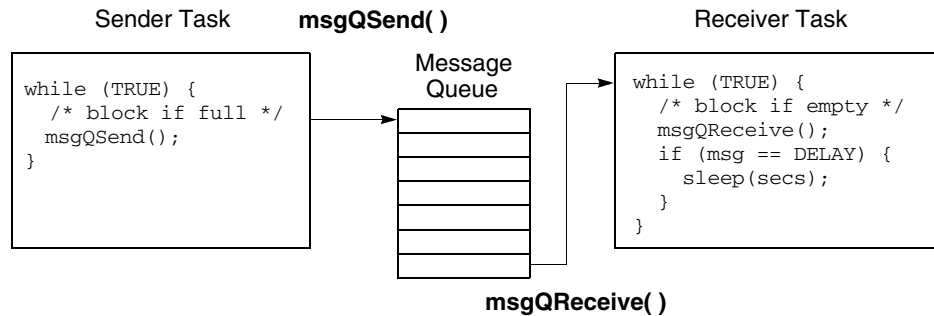
In the **Main** window, select the `malloc(100)` that appears in the `tShell()` column, then open the **Detail** table (see [Detail Table](#), p.61) by selecting the arrow next to **Detail** on the left edge of the window.

You should see a trace record appear in the Function Tracer **Main** window similar to the one below.



Typical Example

Some of the features of Function Tracer can be illustrated using a simple example. This example involves two tasks communicating through a message queue.



The **sender** task sends a series of messages to the queue as fast as it can, and the **receiver** task empties the queue as quickly as it can. When the **receiver** task, however, receives a message that starts with the string **DELAY** it sleeps for the specified number of seconds before resuming. Sleeping causes the message queue to fill, eventually resulting in the **sender** task also blocking.

This example traces the following functions to monitor interaction between the tasks:

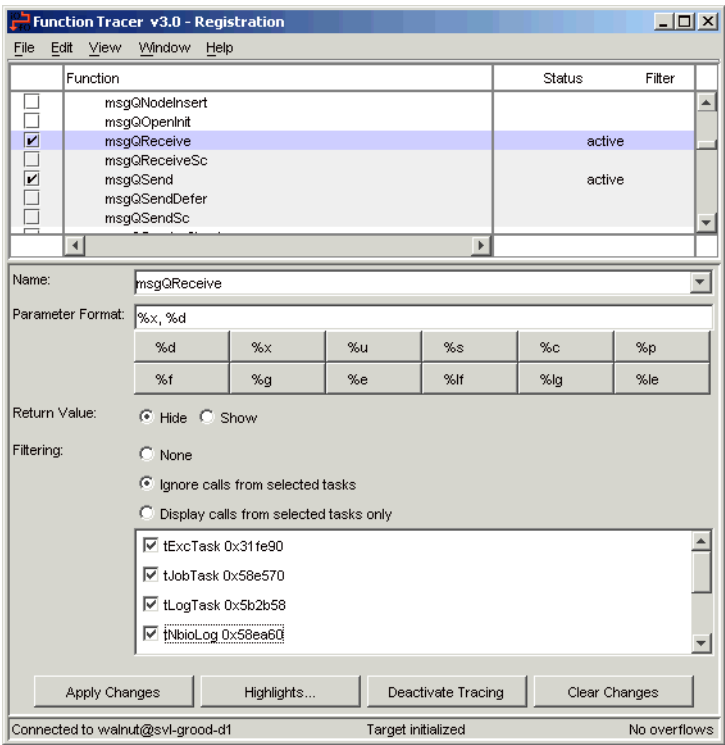
#### **msgQSend()**

This is user function prepares a message before calling the VxWorks function, **msgQSend()**, to send the message. It will block if the message queue is full.

#### **msgQReceive()**

This VxWorks function gets a message from the message queue. The **receiver** task makes the call in blocking mode, so it blocks when the message queue is empty.

The setup for tracing these functions using Function Tracer is shown in the **Registration** window.

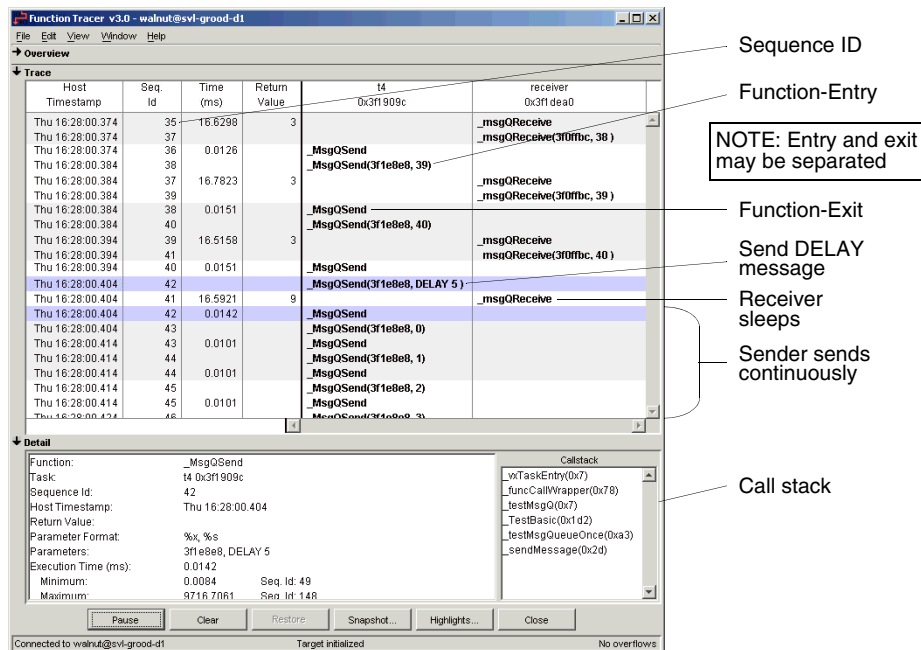


**NOTE:** Since `msgQReceive()` is used by the operating-systems tasks, this example uses task filtering to ignore the following tasks for `msgQReceive()`:

- `tExcTask`
- `tLogTask`
- `tNetTask`
- `tWdbTask`

This figure shows an early portion of a trace collected when the **receiver** task was running at a higher priority than the **sender** task.





From this view you can observe the following events:

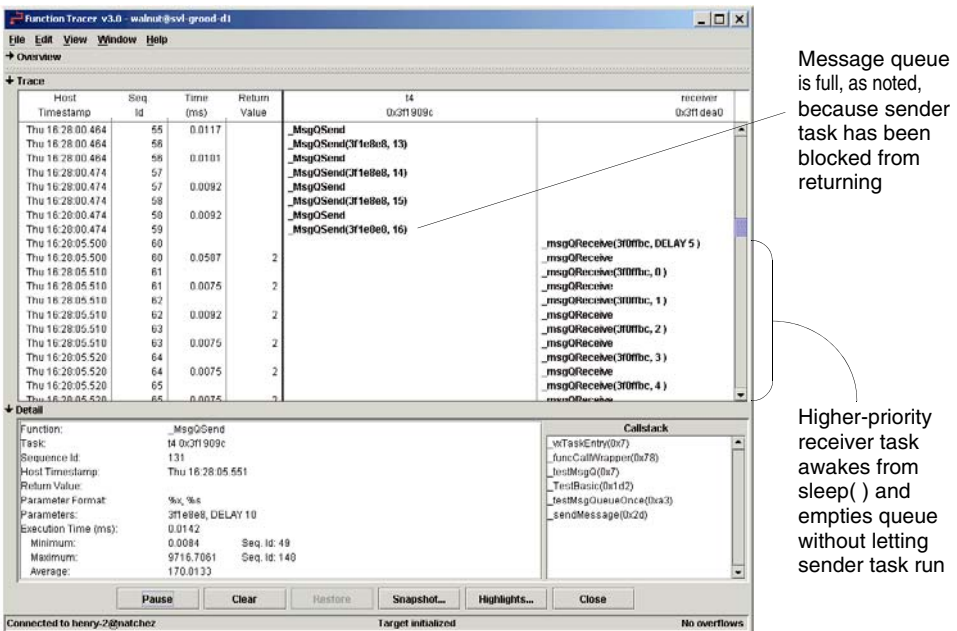
- The calling parameters and return values of each call. Function-entry and function-exit appear as two separate records, sharing the same sequence ID, such as 35, 37, and so forth. The sequence IDs allow you to match function-entry with the corresponding function-exit. Note that sequence ID pairs can be separated by other function calls, but they always occur in pairs—an entry and an exit.
- Observe that when the **receiver** task is not sleeping, it preempts the **sender** task as soon as a message arrives, not even letting the `MsgQSend()` function return.

This causes the call sequences to bounce between the two tasks, as can be seen from the traced sequence IDs 35 through 41.

- Verify that as soon as the **receiver** task receives the **DELAY** message, it sleeps, allowing the **sender** task to send continuously without preemption by the **receiver** task.

- Show from the trace of the **receiver** task that the entry-exit pair for each **msgQReceive()** call is split by an **MsgQSend()** call by the **sender** task. This indicates that the message queue is empty when the **receiver** task calls **msgQReceive()** and that the call completes only after a message has been sent by the **sender** task.
- Show from the trace of the **sender** task that preemption is working properly because the entry and exit for each **MsgQSend()** call is split by a **msgQReceive()** call for the corresponding message from the higher-priority **receiver** task.
- View the call stack of any call by selecting the function-entry record in the table.

This figure shows a slightly later portion of the same trace.



From this view, you can determine the following additional information:

- The **sender** task does indeed block when the message queue fills (sequence ID 59).

- The size of the message queue is the number of uninterrupted **MsgQSend()** calls made by the **sender** task. Referring back to the two previous figures in this section, observe that the first successful call is sequence ID 43 and the last successful call is sequence ID 58, and that, in our example, the message-queue size is 16.
- The **receiver** task empties the message queue before allowing the **sender** task to run again (sequence IDs beginning with 64).

This example shows just some of the information you can obtain from a trace log. It also shows that you can trace your own code as easily as you can operating-system code. Most important of all, the registration of traced functions requires no special compilation, no special link, and no debugging information, allowing you to analyze production code immediately and painlessly.



# 3

## *The Function Tracer GUI*

### 3.1 Introduction 23

### 3.2 The Function Tracer GUI 23

## 3.1 Introduction

The Function Tracer graphical user interface (GUI) is the command center from which you control and view the target program execution traces. This chapter describes each of the GUI elements and its function, with references to its use in displaying target program flow.

## 3.2 The Function Tracer GUI

The Function Tracer GUI comprises the following windows and dialog boxes:

- **Registration** window

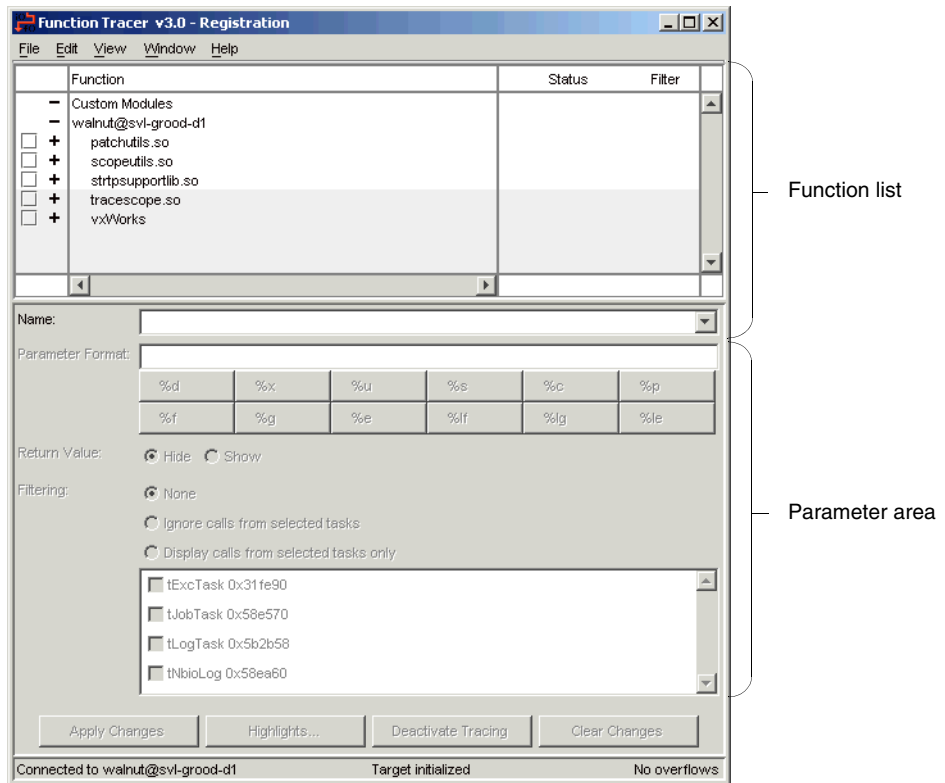
This is where you register trace points to be traced and analyzed (see [3.2.1 Registration Window](#), p.24).

- **Main window**  
Displays the collected and analyzed target code data (see [3.2.2 Main Window](#), p.28).
- **Source Code View window**  
Displays the source code associated with a selected function (see [3.2.3 Source Code View Window](#), p.33).
- **Snapshot window**  
Displays a snapshot, or static view of all the trace records received and stored since Function Tracer was started (see [3.2.4 Snapshot Window](#), p.37).
- **Console window**  
Reports error and warning messages (see [3.2.5 Console Window](#), p.39).
- **Highlight window**  
Configures criteria and colors for highlighting selected trace calls (see [3.2.6 Highlight Window](#), p.40).
- **Columns dialog box**  
Selects the columns you want displayed in the **Overview** and **Trace** tables (see [3.2.7 Columns Dialog Box](#), p.44).
- **Custom Modules dialog box**  
Adds new modules and routines that were not loaded with the target libraries (see [3.2.8 Custom Modules Dialog Box](#), p.45).

These window and dialog boxes are described in detail in the sections that follow.

### 3.2.1 Registration Window

The **Registration** window lets you define all the settings for the current session of Function Tracer, including the registration and activation of trace points. In addition to the **Main** window opening when you first start the GUI, the **Registration** window opens to let you register trace points. You can also open the **Registration** window, or bring it to the top, using the **View > Registration** menu item (see [View](#), p.32).



## Window Elements

The **Registration** window elements are:

- **Title Bar**

The title bar indicates the Function Tracer version and the title **Registration**.

- **Menu Bar**

The menu bar is identical to that for the **Main** window (see [Main Window](#), p.28). Menu items not applicable to this window are grayed out.

- **Function List**

This list contains an entry for each module and function representing the loaded target libraries. The fields are for selecting and activating trace points.

The columns in the **Function** list are:

- (check box)

This check box is used to activate or deactivate tracing for a module or function. When you select the check box, tracing for this module or function begins immediately. When you uncheck the check box, tracing for this module (and all its functions) or individual function is stopped.



---

**NOTE:** If you select the check box for a module, all the function check boxes for that module are checked and activated immediately. If this is not intended, you can select the module check box again to uncheck and deactivate it, and all the function check boxes are also unchecked and immediately deactivated.

---



---

**NOTE:** If you select the check box for a module, and the module contains more than 50 functions, the performance of your target may be adversely impacted. In this case a warning message is issued (see [Activating Trace Points](#), p.54).

---

- **Function**

The list of functions available to be traced. These are taken from the target libraries loaded at startup.

- **Status**

The status of the function. Indicates **Active** if the function is currently being traced; blank otherwise.

- **Filter**

The status of task filtering. Indicates **On** if a Filtering option other than **None** is selected; blank otherwise.

The **Function** list includes the following text field:

- **Name**

Use this text field to enter the name of the function to be located for registration. When you select a trace point from the **Function** list, this field is updated to the name of the selected trace point.



---

**CAUTION:** Be aware that this **Name** matching field only finds the first occurrence of the named function in the **Function** list. If you know, or suspect, that there are multiple occurrences, you must scroll through the list by hand to find the remaining occurrences.

---



- **Parameter Area**

In this area you may enter data for some specific registration parameters. The parameter values displayed are default initially, but always reflect the last selected function after one has been selected.

The parameters are:

- **Parameter Format**

Use this text field to enter the format string for the function parameters. For a description of the formats, see [Registering Trace Points](#), p.51. When you select a trace point from the **Function** list, this field is updated to the parameter format string used by the selected trace point. Use the **Parameter Format** text entry field to build formats with embedded text if desired, selecting valid **printf**-like formats from the format buttons below the list. See examples in [Registering Trace Points](#), p.51.




---

**NOTE:** It is not required that you use the format buttons to enter valid **printf**-like formats; you can type them directly into the **Parameter Format** text entry field.

---

- **Return Value**

When **Show** is selected, Function Tracer prints the return values in the function-exit trace records in the **Trace** table of the **Main** window (see [Return Value Column \(optional\)](#), p.59).

- **Filtering**

This section contains commands to enable or disable task filtering for the trace point that you are adding or modifying.

The filtering commands are:

**None**

Turns off filtering, causing Function Tracer to record all calls to the trace point function.

**Ignore calls from selected tasks**

Ignores trace records for the tasks listed (checked) in the associated task list.

**Display calls from selected tasks only**

Displays trace records for only the tasks listed in the associated task list. (This list is then referred to as a list of watched tasks.)

- **Buttons**

Buttons are located just above the bottom of the window to provide more convenient access to the most frequently used menu bar items.

The buttons are:

- **Apply changes**

When selected, registers a new trace point or commits changes to a currently registered trace point. If currently activated, Function Tracer updates the trace point parameters in real-time.

- **Highlights**

Opens the **Highlight** window where you can select highlight criteria and colors. This button duplicates the **View > Highlights** menu item described in [Menu Bar](#), p.30.

- **Activate/Deactivate Tracing**

If the selected trace point is not active, the button label is **Activate Tracing**. Clicking the button enables tracing for that trace point, and causes the button label to change to **Deactivate Tracing**. Clicking **Deactivate Tracing** disables tracing for that trace point, and causes the button label to change back to **Activate Tracing**.



---

**NOTE:** For the selected trace point, this action duplicates the checking or unchecking of the **Trace** check box in the **Function** list, described above.

---

- **Clear Changes**

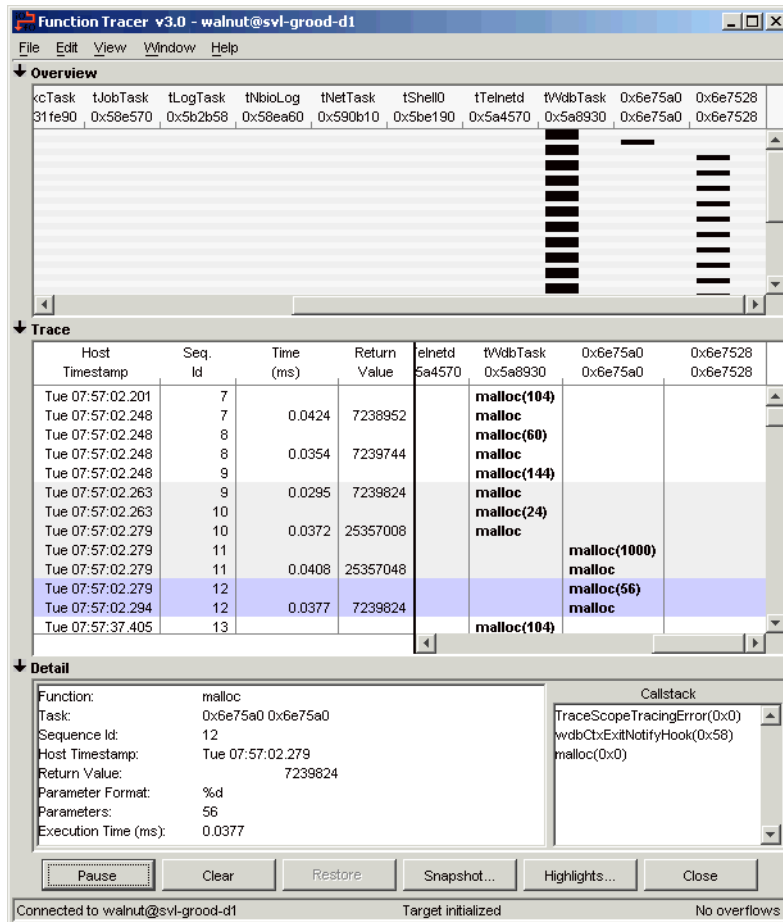
Backs out of any changes made to a trace point before the **Apply Changes** button is clicked.

- **Status Bar**

The status bar appears at the bottom of the **Registration** window. It is identical to the status bar in the **Main** window (see [Status Bar](#), p.65), indicating the current status of the connection with the target.

### 3.2.2 Main Window

The **Main** window displays the collected and analyzed target code data in a set of three user-selectable interpretive tables.



The Main window displays dynamically updated trace records, including timing data, as the trace points are triggered on the target. It also provides access to Function Tracer commands through its toolbar and menus. The window elements are described in the sections that follow.

## Window Elements

The **Main** window contains the following elements:

- **Title Bar**

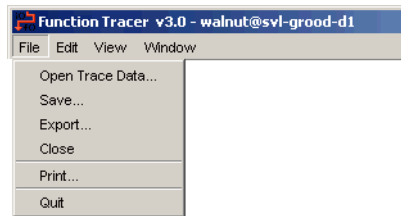
The title bar indicates the version of Function Tracer and the name of the target server to which Function Tracer is connected.

- **Menu Bar**

The menu bar allows access to all Function Tracer functionality through the standard menu items provided. The menu items are:

- **File**

The **File** menu provides access to the file-related commands as well as the **Close** and **Quit** commands.



The **File** menu commands are:

**Open Trace Data**

Opens a binary file, in a separate window, containing previously saved **Trace** table data for examination and analysis.

**Save**

Saves the current contents of the **Trace** table to a binary file (to be opened later with **Open Trace Data**). A **Save** dialog box opens where you can specify the file name.

**Export**

Saves the current contents of the **Trace** table to a file in ASCII format, which may then be printed or imported directly into a spreadsheet application such as MS Excel. A **Save** dialog box opens where you can specify the file name. Files in this format *cannot* be reloaded into Function Tracer with **Open Trace Data** (use the **Save** command for that).

**Close**

Closes the window.

### Print

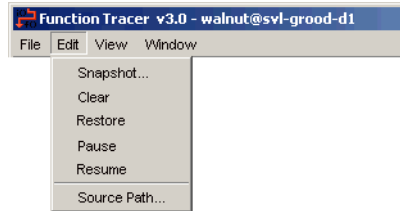
Prints the current log to a printer. A **Print** dialog box opens where you can select printing parameters.

### Quit

Quits Function Tracer. Quitting also de-initializes the target libraries, releasing any target memory used for tracing.

### – Edit

The **Edit** menu allows you to modify the display of data items on the screen.



The **Edit** menu commands are:

### Snapshot

Copies the entire contents of the **Trace** table into a separate window ([3.2.4 Snapshot Window](#), p.37). The contents of the newly created **Snapshot** window are no longer updated with fresh data.

### Clear

Clears the **Overview**, **Trace**, and **Detail** tables in the **Main** window. You may want to do this after saving the log to reduce memory usage by the GUI and to improve performance.

### Restore

Restores all data in the **Overview** and **Trace** tables removed with **Clear** since the beginning of the current trace session. It does not restore any data previously displayed in the **Detail** table.

### Pause

Temporarily stops the display only of trace data being generated. When the trace display is in the paused state, **Pause** is grayed out.

### Resume

Restarts the display of trace data that was stopped with **Pause**. When the trace display is not in the paused state, **Resume** is grayed out.

### Source Path

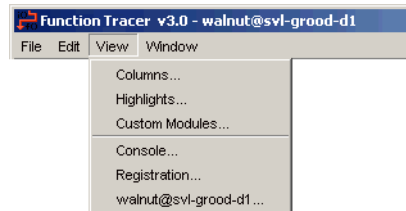
Opens the Source Path dialog box (see [Source Path Dialog Box](#), p.35) where you can specify directory paths to search for source code display.



**NOTE:** In the **Source Path** dialog box, the vertical scroll bar on the right may move itself upward with the influx of data, which inhibits the continuing display of generated records. If this happens, simply move the vertical scroll bar to the bottom and dynamic data display resumes.

### – View

Use the **View** menu to access the view-related commands, including opening the **Registration** window.



The **View** menu commands are:

### Columns

Opens the **Columns** dialog box ([3.2.7 Columns Dialog Box](#), p.44), where you can select which columns (tasks) to hide or display trace records for.



**NOTE:** Tasks hidden with this command continue to store any generated trace records. If turned on later, all trace data stored while it was hidden are displayed.

### Highlights

Opens the **Highlight** window ([3.2.6 Highlight Window](#), p.40), where you can select highlight criteria and colors to be applied to generated trace records.

### Custom Modules

Opens the **Custom Modules** dialog box ([3.2.8 Custom Modules Dialog Box](#), p.45), where you can add modules and functions not initially loaded with the target libraries.

### Console

Opens the **Console** window ([3.2.5 Console Window](#), p.39), displaying error and warning messages. If the **Console** window is opened already, but hidden behind other windows, you can use this menu item to bring the window to the top.

### Registration

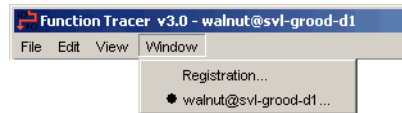
Opens the **Registration** window([3.2.1 Registration Window](#), p.24), where you select target functions for tracing and set up their parameters.

### *targetServer*

Opens the target server (**Main**) window, where *targetServer* is the name of the target server.

### – Window

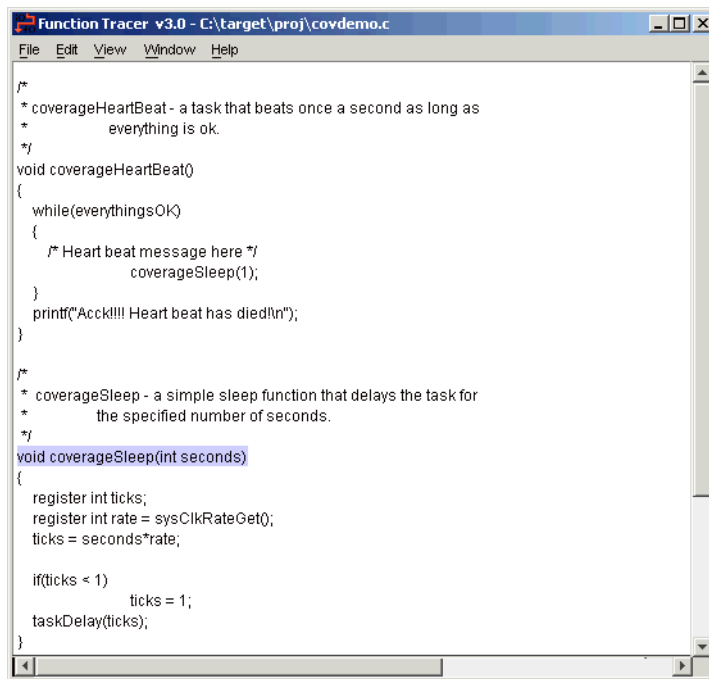
The **Window** menu lists all the windows currently open.



It changes dynamically as you open and close windows. You can select any window listed in the menu to bring it to the top if it is hidden beneath other open windows.

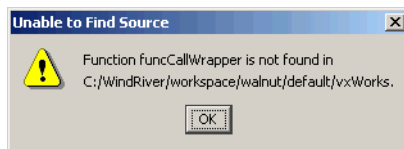
## 3.2.3 Source Code View Window

The **Source Code View** window allows you to view the source code associated with a selected function. An example of a Source Code View window is shown here.



This window is opened by selecting the function name in the **Callstack** list displayed in the **Detail** table in the **Main** window (see [Detail Table](#), p.61). The displayed source code cannot be edited; it is only available for viewing.

In the event the source code file for the selected function is not found, the **File Not Found** error dialog box is displayed.

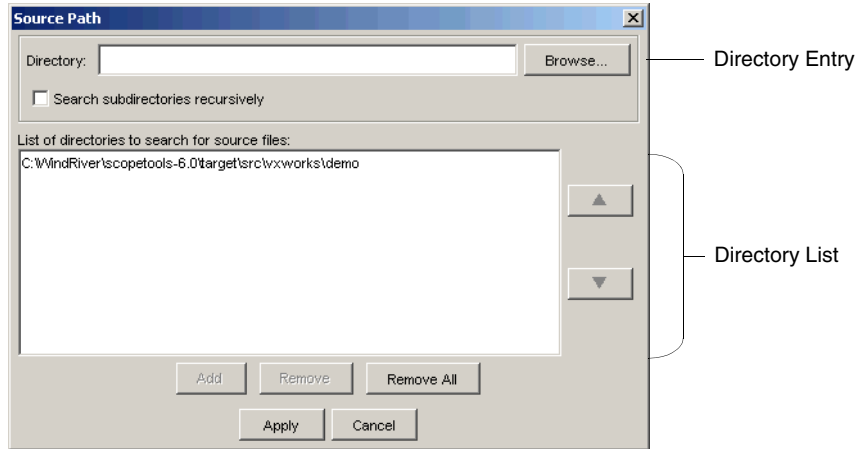


No action can be taken from this dialog box; click **OK** to dismiss it. Then select **Edit > Source Path** from the menu bar to open the **Source Path** dialog box (described in the next section) where you can enter the correct directory to search.



## Source Path Dialog Box

Open the **Source Path** dialog box using the **Edit > Source Path** menu command.



You can enter an ordered list of directories here to search for source code files.

## Window Elements

The **Source Path** dialog box screen includes the following elements:

- **Directory Entry Panel**

A set of controls for editing and reordering the directory list, containing the following items:

- **Directory**

- A text field where the full path to a directory can be manually entered.

- **Browse**

- A button that opens a file browser to graphically select directories.

- **Search subdirectories recursively**

- Select this check box to search each listed directory recursively.

- **List of directories to search for source files**

An ordered list of directories searched when trying to find a source code file. The list is searched in order from top to bottom. To change the search priority of any entry, select the row, then use the **Move Up** or **Move Down** arrow

buttons to physically move the row up or down in the table with respect to the other rows.

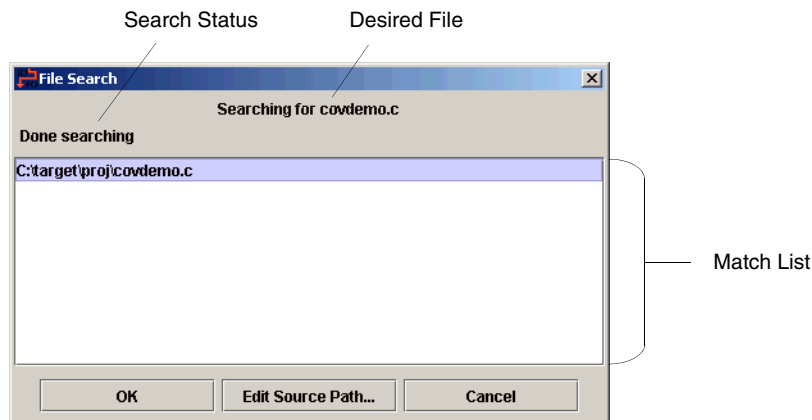
▪ **Buttons**

The following buttons are provided:

- **Add**  
Adds the contents of the directory text field to the bottom of the list.
- **Remove**  
Removes the selected directory from the list.
- **Remove All**  
Removes all directories from the list.
- **Apply**  
Saves the changes made to the directory list, closes the **Source Path** dialog box and opens the **File Search** dialog box (below).
- **Cancel**  
Closes the dialog box without saving any changes you made.

## File Search Dialog Box

The **File Search** dialog box is displayed while the software is actively searching the list of specified directories for a source code file.



Searching for a source code file in a large number of directories can be time consuming. Any matching files found are added to the match list and can be selected at any time, even while the search is continuing to run. Once a match is found and selected, the **OK** button is enabled. Clicking any of the other buttons at the bottom while the search is in progress stops the search immediately.

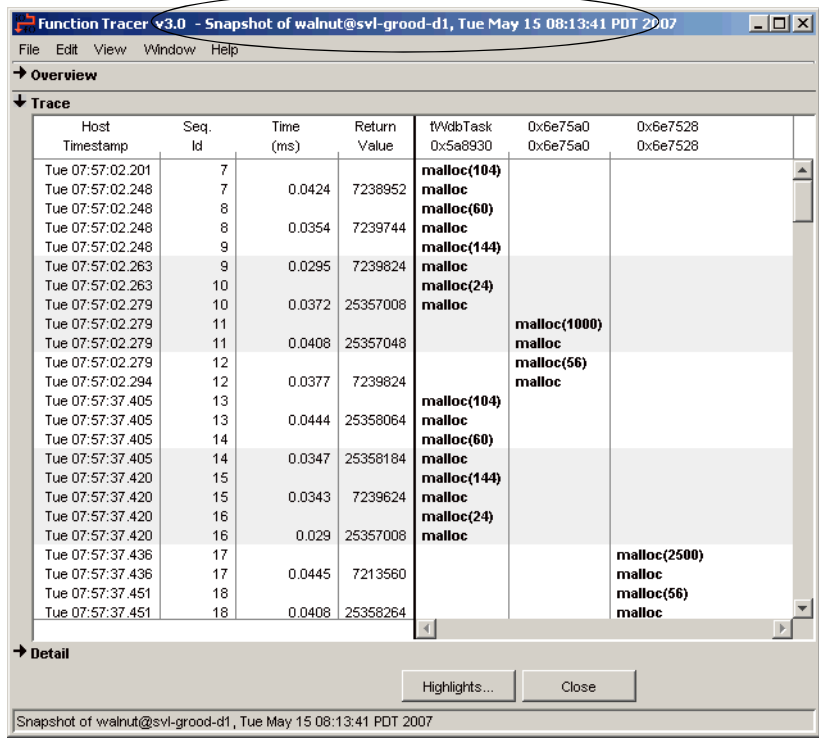
### Window Elements

The **File Search** dialog box has the following screen elements:

- **Desired File**  
The name of the source code file being searched for.
- **Search Status**  
A field containing either the name of the directory currently being searched or, if finished, the text **Done searching**.
- **Match List**  
The list of files whose name matches that of the desired source code file.
- **Buttons**  
The following buttons are provided:
  - **OK**  
Becomes enabled when a file is selected in the match list. It closes the dialog box and opens the selected file in a **Source Code View** window, described in [3.2.3 Source Code View Window](#), p.33.
  - **Edit Source Path**  
Closes the dialog box and opens the **Source Path** dialog box, described in [Source Path Dialog Box](#), p.35.
  - **Cancel**  
Closes the dialog box.

### 3.2.4 Snapshot Window

The **Snapshot** window displays a static view of all the trace records received and stored since your target program was started.



These records consist of all the entries displayed in the **Main** window (described in [3.2.2 Main Window](#), p.28) up to the instant the snapshot was taken. A snapshot is denoted by the word **Snapshot**, along with the filename, date and time of the snapshot, displayed in the title bar (circled above).

The task columns shown in the **Snapshot** window above are only the ones containing trace data actually collected; empty task columns are not displayed. The records in this window are static, and are no longer updated.

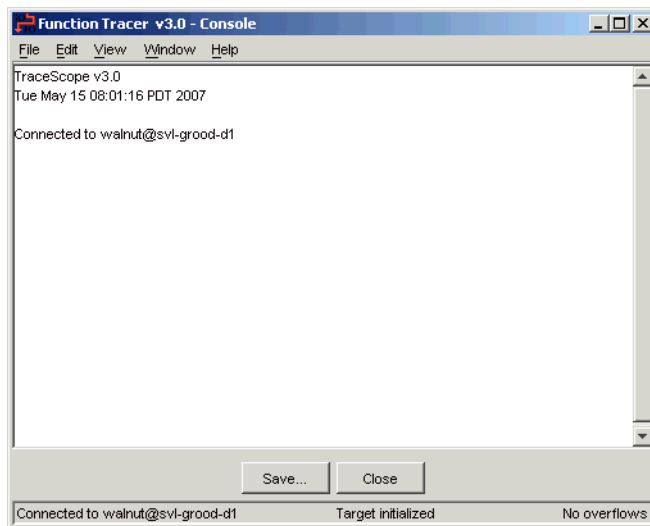
A snapshot window is created by using the **Edit > Snapshot** menu command. Any number of **Snapshot** windows may be opened simultaneously to allow you to perform comparisons of traced data. The **Snapshot** window appears and operates nearly identically to the **Main** window (see [3.2.2 Main Window](#), p.28), including the menu, toolbar, and buttons.



**NOTE:** A snapshot is a temporary window. If you want a permanent record, you must use the **File > Save** menu command (in the Snapshot window) to save it to a file (see [Menu Bar](#), p.30).

### 3.2.5 Console Window

The **Console** window is where Function Tracer reports errors and warnings. Open it using the **View > Console** menu command in the **Main** window.



During normal operation, very few messages are printed to the **Console** window. However, when you start Function Tracer with a non-zero verbosity level (see [2.3 Starting Function Tracer](#), p.9), the amount of output can be significant.



**NOTE:** Read the Warning note in [Starting Function Tracer From Workbench](#), p.11.

### Window Elements

The **Console** window elements are:

**Title Bar**

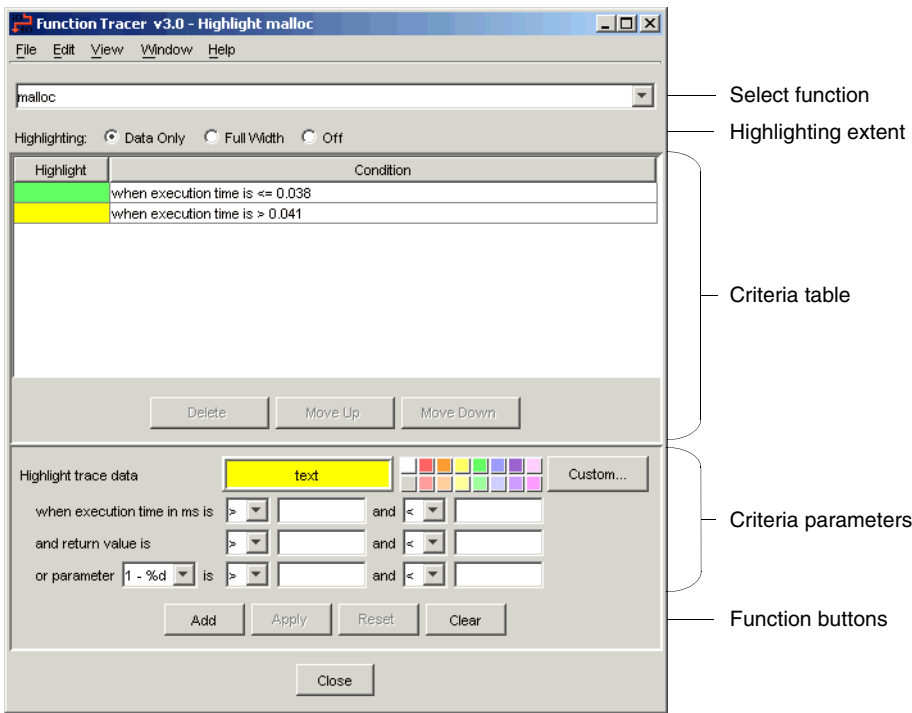
Indicates the version of Function Tracer and the title **Console**.

**Menu Bar**

The menu bar for the **Console** window is identical to that for the **Main** window (*Menu Bar*, p.30). Menu items not applicable to this window are grayed out.

**3.2.6 Highlight Window**

You can configure criteria and colors for highlighting selected trace calls using the **Highlight** window.



Highlighting selected trace records with different colors as they are displayed aids in spotting and analyzing performance characteristics of the target system. Highlight criteria can be applied before tracing begins, and can also be applied and

modified as tracing progresses. All additions and modifications take place immediately, and remain until changed again.

The window may be opened by either of the following actions:

- Clicking the **Highlights** button in the **Main** window or the **Registration** window.
- Selecting the **View > Highlights** menu command.

## Window Elements

The **Highlight** window elements are:

### Title Bar

Indicates the version of Function Tracer, the title **Highlight**, and the name of the selected function you are highlighting.

### Menu Bar

The menu bar for the **Highlight** window is identical to that for the **Main** window (*Menu Bar*, p.30). Menu items not applicable to this window are grayed out.

### Function Selection Field

Allows you to select or enter the name of the function.

### Highlighting

These buttons allow you to determine how much of the trace line gets colored.

### Criteria Table

This table displays the highlighting parameters for each configured function.

### Criteria Parameters

Individual parameter entry fields allow you to select parameter types and limits for highlighting.

### Function Buttons

Use these buttons to manipulate the criteria selections made.

### Close Button

Closes the window.

## Configuring Highlight Criteria

You can configure, or modify existing configurations for, highlighting trace records at any time using the **Highlight** window (see the previous figure in [3.2.6 Highlight Window](#), p.40).

1. Locate the function name (trace point) in the drop-down list, or enter the name of the function in the function selection field. The highlighting criteria specified is applied to, or modified in, only this trace point.
2. **Select Data Only** (highlights only the data cell), **Full Width** (highlights the entire line), or **Off** (suspends highlighting without changing the criteria) for Highlighting.
3. Select a color from the **Highlight** trace data field for the selected trace point. You may create a new color by clicking the **Custom** button.
4. Select the parameter types and limits using the three following sets of text entry boxes:

### **when execution time in ms is...and**

Select a boolean symbol for the first text box from the drop-down list, then enter the time (in milliseconds) in the first text box. If you want to bracket a time, select the appropriate symbol for that, and enter the second time in the second text box.

### **and return value is...and**

Select return value(s) in the same way, to be logically "**and**"ed with the execution time selected above, if any.

### **or parameter...is...and**

Select a passed input value in the same way, except first select the parameter format number from the drop-down list preceding the two text entry boxes.

5. Click one of the following function buttons to finalize the trace point criteria selection:

### **Add**

Add this new trace point, with the selected criteria, to the **Criteria** table. The trace point appears in the table.

### **Apply**

Apply the above criteria to the selected trace point. Changes to the trace point criteria appear in the table.



### **Reset**

Causes any changes made to criteria for a selected trace point, before clicking **Apply**, to be returned to their previous value.

### **Clear**

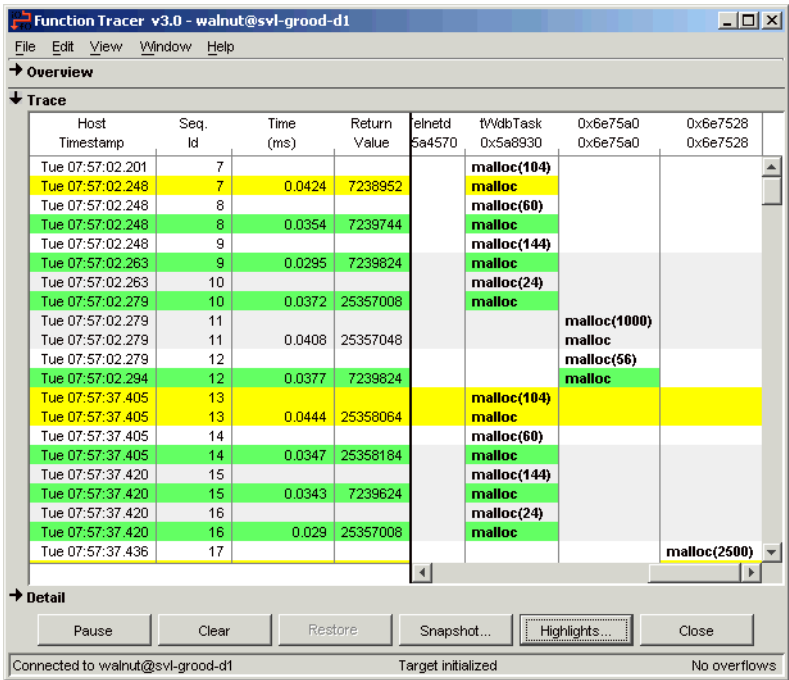
Clears all values set in the text entry fields. You must click **Apply** to save the blank fields.

6. Modify the priority of the highlight criteria for the function by physically moving a selected criteria up (higher priority) or down (lower priority) in the criteria table using the **Move Up** or **Move Down** buttons at the bottom of this table. You can delete all criteria from a function using the **Delete** button.
7. Click **Close** to close the window.

All highlight criteria set for a function remains, even across Function Tracer sessions, until modified or deleted.

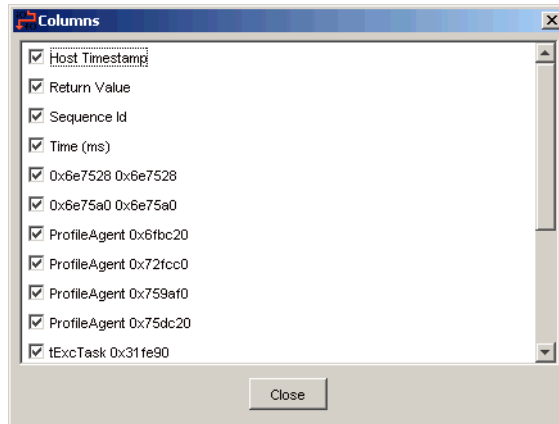
### **Example**

An example of highlights generated from the setup in the **Highlight** window shown in [3.2.6 Highlight Window](#), p.40 is shown here.



### 3.2.7 Columns Dialog Box

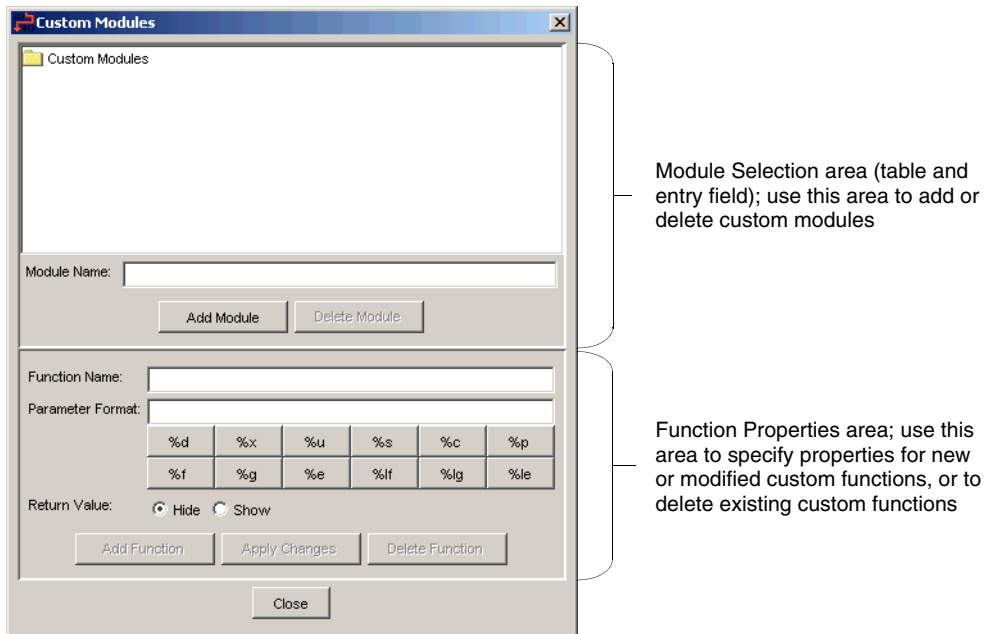
Use the **Columns** dialog box, opened with the **View > Columns** menu command, to select the columns you want displayed in the **Overview** and **Trace** tables in the **Main** window (in the live window as well as the **Snapshot** window).



Select the check boxes for each column you want displayed in the **Main** window (or in the **Snapshot** window). The result of excluding or including columns appears immediately, but only affects the window in which they are selected, and only for the current session.

### 3.2.8 Custom Modules Dialog Box

The **Custom Modules** dialog box is where you can add new modules and routines that were not loaded with the target libraries, or are not otherwise known to the target server. Open this dialog box using the **View > Custom Modules** menu command on the **Main** window or **Registration** window menu bar.



## Window Elements

The **File Search** dialog box contains the following window elements:

- **Module Selection Panel**  
Where you add or delete custom modules. The **Module Selection Panel** contains the following items:
  - **Custom Modules table**  
The list of custom modules and functions that you create.
  - **Module Name**  
The text entry field where you enter the new module name.
  - **Buttons**  
The following buttons apply to the **Module Selection Panel**:
    - Add Module**  
Adds the new module in the **Module Name** field to the **Module** table.

### Delete Module

Deletes a selected module from the **Module** table.

- **Function Properties Area**

Where you select and enter properties for a new or modified function to be added to the module. It contains the following items:

- **Function Name**  
The text entry field where you enter the new or modified function name.
- **Parameter Format**  
Select the parameter formats for the function using the buttons below the field, or type the formats directly into the field.
- **Return Value**  
Select the **Hide** or **Show** button to hide or display the return value.
- **Buttons**  
The following buttons apply to the **Function Properties Panel**:

#### Add Function

Adds the new function to the **Module** table.

#### Apply Changes

Applies any changes you make to a selected function.

#### Delete Function

Deletes the selected function from the **Module** table.

- **Buttons**

The following buttons apply to the **File Search** dialog box overall:

- **Close**  
Closes the dialog box.

## Loading Custom Modules

You can begin loading custom modules and routines at any time after Function Tracer has finished loading target libraries using the **Custom Modules** dialog box (described above).

1. Enter a name for your new module in the **Module Name** field, or select a custom module name to modify from the table in the upper (**Module Selection**) panel of the window.

2. If this is a new module, click **Add Module** to add it to the **Custom Modules** table in the upper panel. You can also delete a selected module, along with all its functions, from the folder by clicking **Delete Module**.
3. Enter a new custom function name in the **Function Name** field, or select a function from a module in the **Custom Modules** table for modification.
4. Use the **Parameter Format** text field to enter the format string for the function parameters. For a detailed description of the formats, see [Registering Trace Points](#), p.51. This format string defines the input parameters to the new or modified function.



---

**NOTE:** It is not required that you use the format buttons to enter valid **printf**-like formats; you can type them directly into the **Parameter Format** text entry field.

---

5. When **Show** is selected for **Return Value**, Function Tracer displays the return value in the function-exit trace records in the **Trace** table of the **Main** window (see [Return Value Column \(optional\)](#), p.59). Otherwise it is not displayed.
6. Click one of the following buttons to finalize the custom function description:

**Add Function**

Add this new function to the table. The function appears in the table.

**Apply Changes**

Apply selected values to the selected existing function being modified.

**Delete Function**

Deletes the selected function from the module.

7. Click **Close** to close the **Custom Modules** dialog box.

The module(s) and function(s) created with this procedure can now be registered for tracing (see [Registering Trace Points](#), p.51).

# 4

## *Using Function Tracer*

- 4.1 Introduction 49
- 4.2 Starting Tracing Activity 49
- 4.3 Viewing Data 56
- 4.4 Operational Features 66
- 4.5 System Viewer Event Integration 68

### 4.1 Introduction

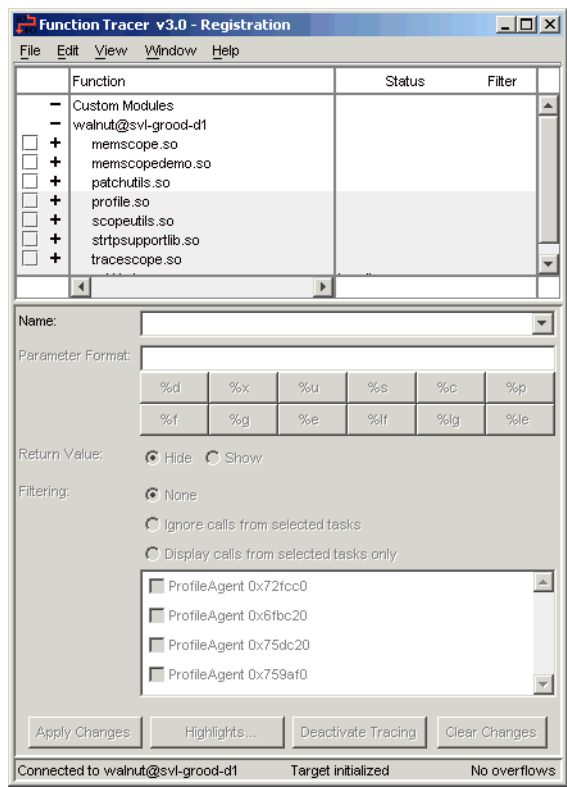
This chapter goes into the details of using the Function Tracer GUI to analyze and display real-time data from your running target program. It describes the startup of Function Tracer, and the subsequent control of data flow and analysis using all the GUI features.

### 4.2 Starting Tracing Activity

For information on how to launch the GUI, see [2. Getting Started](#).

Initializing Trace Points

The **Registration** window appears when you first start the GUI.



This window is where you define all the settings for the current session of Function Tracer, including the registration and activation of trace points. It opens automatically at startup, but you can also open it at any time using the **View > Registration** menu item (see [View](#), p.32).

Before any data can be displayed, you must register and activate at lease one trace point in this window. The process of managing trace points for routines and functions you want to track is described in the sections that follow.

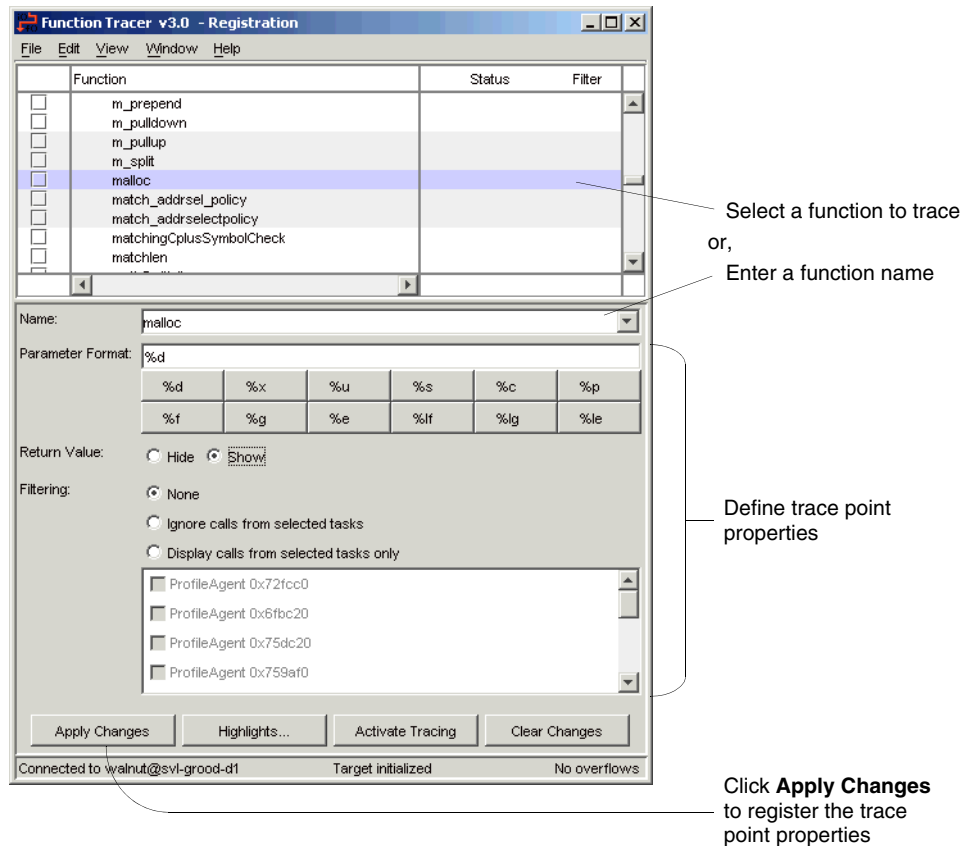


## Registering Trace Points

Registering a trace point initializes the trace point settings for a function, but the Function Tracer target agent does not start tracing the function until the trace point is **activated**. Therefore, you can register many functions in the beginning, but only activate a few at a time to minimize the impact on your running programs. The settings used this time are saved and reinstated the next time you start Function Tracer.

Register a trace point by following these steps:

1. In the **Registration** window, double-click the name of a module in which you want to trace the functions. This opens the list of functions for that module.



For details, see [3.2.1 Registration Window](#), p.24.

2. Within Function Tracer, a trace point is identified by a function name, but the trace point is more than just the function name. It is composed of the following specifications, applied to that function, that you must make during registration.

- a. Choose a function to trace by selecting its name in the list, or by entering its name in the **Name** field. Since many C compilers create target symbols by prepending an underscore ("\_") to the name of a function, you may also need to prepend the underscore when searching for the function to trace using the **Name** field.
- b. Enter the format string for the function parameters, if any, in the **Parameter format** field. The supported formats are:

**character %c**

A one-byte character.

**decimal int %d**

An integer in decimal format.

**double %lg**

A double-precision number printed in general format that adjusts the precision of the output automatically, resorting to exponential format when needed.

**double %lf**

A double-precision number printed in non-exponential format.

**double %le**

A double-precision number printed in exponential format.

**float %g**

A single-precision number printed in general format that adjusts the precision of the output automatically, resorting to exponential format when needed.

**float %f**

A single-precision number printed in non-exponential format.

**float %e**

A single-precision number printed in exponential format.

**hex int %x**

An integer printed as a hexadecimal number (without the leading 0x).

**pointer %p**

A pointer printed as a hexadecimal number (whether the leading 0x is printed is compiler dependent).

**string %s**

A string; a pointer to a null-terminated array of characters to be printed as a string. The string should not contain new-line characters.

**unsigned %u**

An unsigned integer printed in decimal format.



---

**NOTE:** You may omit function parameters from the format string, and Function Tracer just ignores them. However, you should not specify more parameters than the function has.

---



---

**NOTE:** The actual number of parameters printed is limited by the initialization settings (see [Starting Function Tracer From Workbench](#), p.11).

---

- c. Select **Show** in the **Return Value** field if you want Function Tracer to print the return value for the function; otherwise select **Hide**.
- d. Select a **Task Filtering** option from:

**None**

Use this setting to record all calls to the function.

**Ignore calls from selected tasks**

Use this setting to define a list of tasks to ignore when recording calls to the function.

**Display calls from selected tasks only**

Use this setting to record calls to the function *only* if the calling task is in the defined list (watched tasks).

If you selected the **Ignore calls from selected tasks** or the **Display calls from selected tasks only** option, define a task list in the associated list box by selecting the check box next to the name of each task you want included.

- 3. Click **Apply Changes** to register the trace point.



---

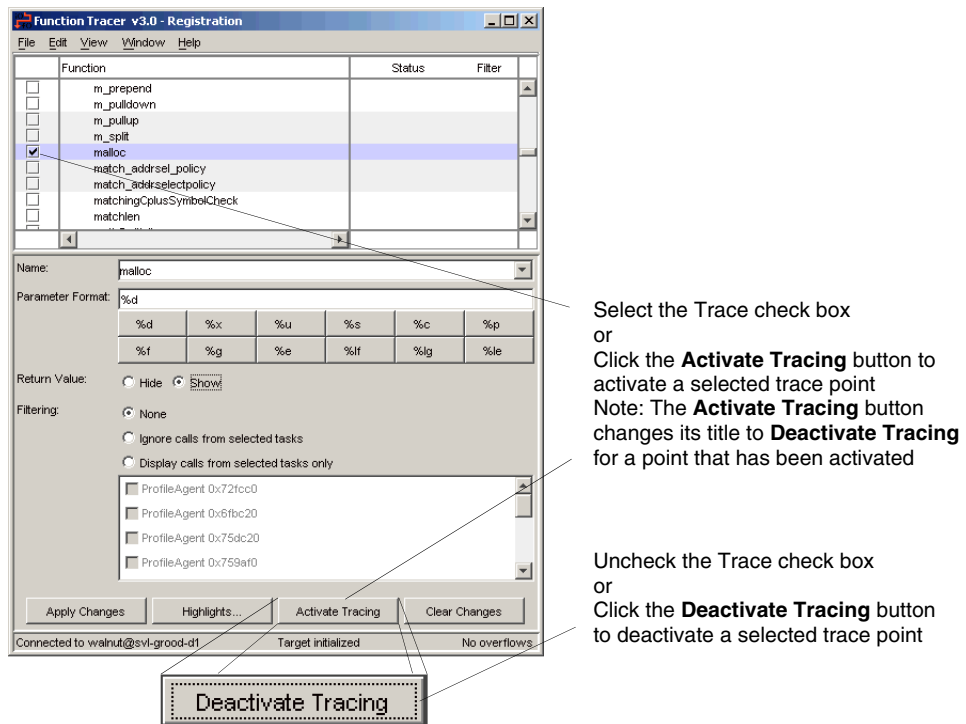
**NOTE:** Do this step, even if you accepted all the default values for this function.

---

4. Select the **Trace** check box if you want to activate the selected trace point immediately after registering it; otherwise register more trace points first.

## Activating Trace Points

You must now activate a trace point before Function Tracer will record calls to the corresponding function. Activate each registered trace point individually using the **Registration** window, as shown here.



In the **Registration** window, select the **Trace** check box for each function you want to activate (or select **Activate Tracing** at the bottom of the window for a selected function).



---

**NOTE:** Activating more than five functions at the same time may result in a significant impact on your running system. Prior to allowing this, a warning message is displayed.

---



---

**CAUTION:** A maximum of 32 trace points is allowed. Most systems can experience significant delays with fewer than 32, so adjust the number according to the maximum processing load or delay your system can tolerate.

---

## Deactivating

Trace points are deactivated individually using the **Registration** window, shown above. In the **Registration** window, uncheck the **Trace** check box for each function you want to deactivate (or click **Deactivate Tracing** at the bottom of the window for each selected function).

## Modifying

You can change the tracing characteristics of any trace point at any time, even while the trace point is activated, using the **Registration** window. Referring to the figure above, follow these steps:

1. In the **Registration** window, locate the traced function in the **Function** list either by scrolling, or by entering its name in the **Name** field.
2. Select the function in the list to copy its tracing characteristics into the appropriate fields in the **Registration** window.
3. Make any desired changes to the name, parameter format, return value, or taskfilter settings.
4. Click **Apply Changes** to update the trace point with the new properties.



---


**NOTE:** If you are actively tracing this function, be sure its check box is still checked for tracing in the **Registration** window. The effects of your changes appear immediately in the **Trace** table ([Trace Table](#), p.58).

---

## Removing

Remove a trace point by deactivating it (see [Deactivating](#), p.55). When you deactivate a trace point, the code patch used to trace and print the activity for that function is removed from the target and no longer influences processing. When you activate the trace point again, the code is reinstated. The registration parameters set for this trace point remain intact until changed.

## 4.3 Viewing Data

The Function Tracer **Main** window is the primary view into the execution of your program. It is where collected and analyzed tracepoint data is displayed in a set of 3 user-selected tables. Each table can be individually opened or closed using the arrow symbol (  ) along the left window margin. The display can be augmented by highlighting data rows of particular interest with different colored backgrounds (see [3.2.6 Highlight Window](#), p.40), or selecting only the columns of data you are interested in viewing (see [3.2.7 Columns Dialog Box](#), p.44).

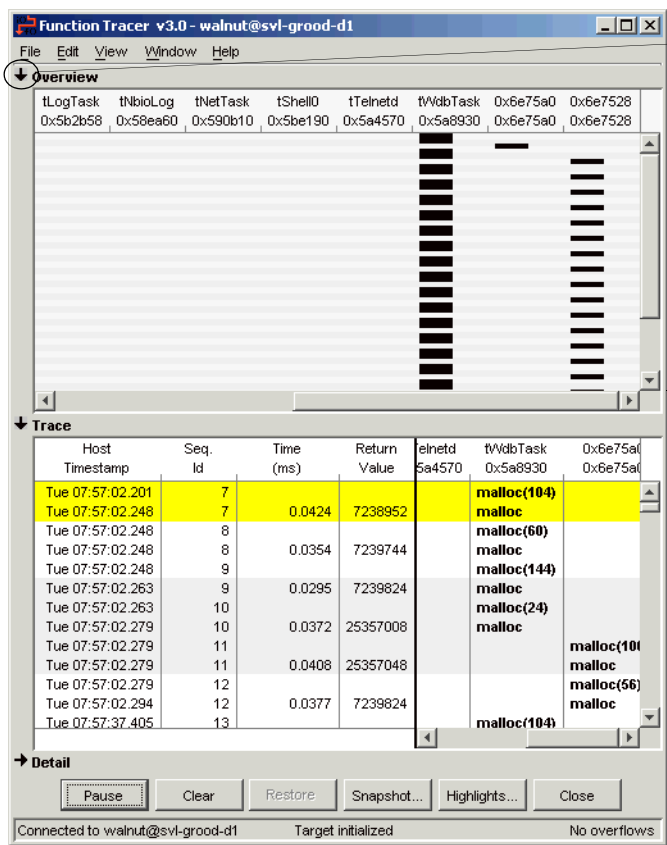
The Main window displays the following tables:

- **Overview**
- **Trace**
- **Detail**

These tables all appear simultaneously in the same (Main) window, and any or all of these tables may be open for view at the same time.

### Overview Table

The **Overview** table (in the **Main** window) is initially hidden, but it can be displayed by toggling the small arrow to the left of the name, or by double-clicking the **Overview** name itself.



Toggle the arrow (->) or double-click **Overview** to open or close the Overview table

Overview table

4

This table displays the tasks by columns, with a scrolling bar graph below it. Each row in the graph represents, in chronological order, an entry to or exit from one of the tasks.

The traced function data appears in columns representing the calling task, with each task in a separate column. They are intended to show only a graphical representation of the CPU activity distribution among the tasks. If the number of tasks is large, viewing the traces could require a lot of scrolling between columns. The selection of displayed task columns can be modified to avoid this.

For details, see [3.2.7 Columns Dialog Box](#), p.44.

In the **Overview** table, you can rearrange the order of the task columns to more effectively visualize the trace results. Do this by following the procedure outlined in [Task Column](#), p.60.



**NOTE:** Any task column location change you make in either the **Overview** or **Trace** table is duplicated in the other table.

Trace Table

The **Trace** table is the main trace data display area. It is initially displayed alone in the **Main** window, but may be hidden or displayed at any time by toggling the small arrow to the left of the name, or by double-clicking the **Trace** name itself.

Host Timestamp	Seq. Id	Time (ms)	Return Value	elnetid	tV/dltTask	0x6e75a0
Tue 07:57:02.201	7			5a4570	malloc(104)	
Tue 07:57:02.248	7	0.0424	7238952		malloc	
Tue 07:57:02.248	8				malloc(60)	
Tue 07:57:02.248	8	0.0354	7239744		malloc	
Tue 07:57:02.248	9				malloc(144)	
Tue 07:57:02.263	9	0.0295	7239824		malloc	
Tue 07:57:02.263	10				malloc(24)	
Tue 07:57:02.279	10	0.0372	25357008		malloc	
Tue 07:57:02.279	11				malloc(104)	
Tue 07:57:02.279	11	0.0408	25357048		malloc	
Tue 07:57:02.279	12				malloc(56)	
Tue 07:57:02.294	12	0.0377	7239824		malloc	
Tue 07:57:37.405	13				malloc(104)	

The **Trace** table dynamically displays all the data records for activated trace points as they are collected from the target. Each row contains a trace record representing an **entry** to, or an **exit** from, a registered function.

The traced function data appears in columns representing the calling task, with each task in a separate column. The selection of displayed columns can be modified as described in [3.2.7 Columns Dialog Box](#), p.44. You can also rearrange the order of the task columns to more effectively visualize the trace results, as outlined in [Task Column](#), p.60.



The **Trace** table elements are described in the following sections.

### Host Timestamp Column

This column is the date and time the trace sample was taken. It is derived from the host computer time clock, and therefore cannot be relied upon for precision timing issues.

For more details, see [Unresolved Symbols at Startup](#), p.13.

### Sequence ID Column

Function Tracer numbers each entry/exit pair of trace records sequentially, starting with 0. The exit point of a traced function call has the same sequence ID number as the entry point. When the traced function calls become nested or the call sequences become more complicated, you can use these sequence IDs to help you match the entry and exit points. For example, if **foo( )** calls **bar( )**, the entries within a column might look like:

```
0 foo(3)
1 bar(1)
1 bar = 0x00000000
0 foo = 0x00000001
```

In this example, **foo( )** uses sequence ID 0 and **bar( )** uses sequence ID 1.

In this column, the sequence ID numbers increase with every trace record, and are not reset to 0 until you quit and restart Function Tracer, or until the target is rebooted.

### Time (ms) Column

If your target has initialized the high resolution timestamp driver, Function Tracer prints, in this column, the approximate amount of time (in milliseconds) it took to execute the traced function. For a description of the driver, see [2.3 Starting Function Tracer](#), p.9.

### Return Value Column (optional)

This column contains the return value from the function call, expressed in both decimal and hexadecimal. Display of the return value may be disabled on a per trace point basis during trace point registration (see [3.2.1 Registration Window](#), p.24).

## Task Column

Each **Task** column corresponds to a task running on the target. All entries within the column represent traced function calls made by that task. The column heading contains the task information **task name/task id**. If a task is short-lived, Function Tracer may not be able to find its name, in which case the task id is substituted for the task name.

For each task column, you can:

- Enable or disable its display by toggling the task name in the **Columns** dialog box ([3.2.7 Columns Dialog Box](#), p.44).
- Physically move the column to a different location by placing the cursor on the column heading, then pressing the left mouse button and dragging it (horizontally) to a new location. When you release the mouse button, the column remains there until you move it again.



---

**NOTE:** Any task column location change you make in either the **Trace** or **Overview** table is duplicated in the other table.

---

## Function-Entry Row

Each function-entry record is a row in the table, and has the following format:

```
funcName (arg1, arg2, ...)
```

where:

### funcName

The name of the function being called. The function name is followed by a pair of parentheses that enclose the function parameters.

### arg1, arg2, ... (optional)

The function parameters passed to the function call are printed using the format specified in the trace point registration. This display of function parameters may be enabled or disabled in the **Parameter format** field of the **Registration** window (see [3.2.1 Registration Window](#), p.24).

When you select a row that contains a function-entry record, the call stack for the function call is displayed in the **Callstack** list of the **Detail** table ([Detail Table](#), p.61).

## Function-Exit Row

Each function-exit record is also a row in the table, and has the following format:

```
funcName
```

If **Show** is selected for **Return Value** for this function in the **Registration** window, the **Return Value** column displays the value, in decimal.

### Scroll Bars

As the number of trace records and the number of tasks grow, the entries may not fit within the viewable portion of the **Trace** table. Use the vertical and horizontal scroll bars to view other parts of the table.



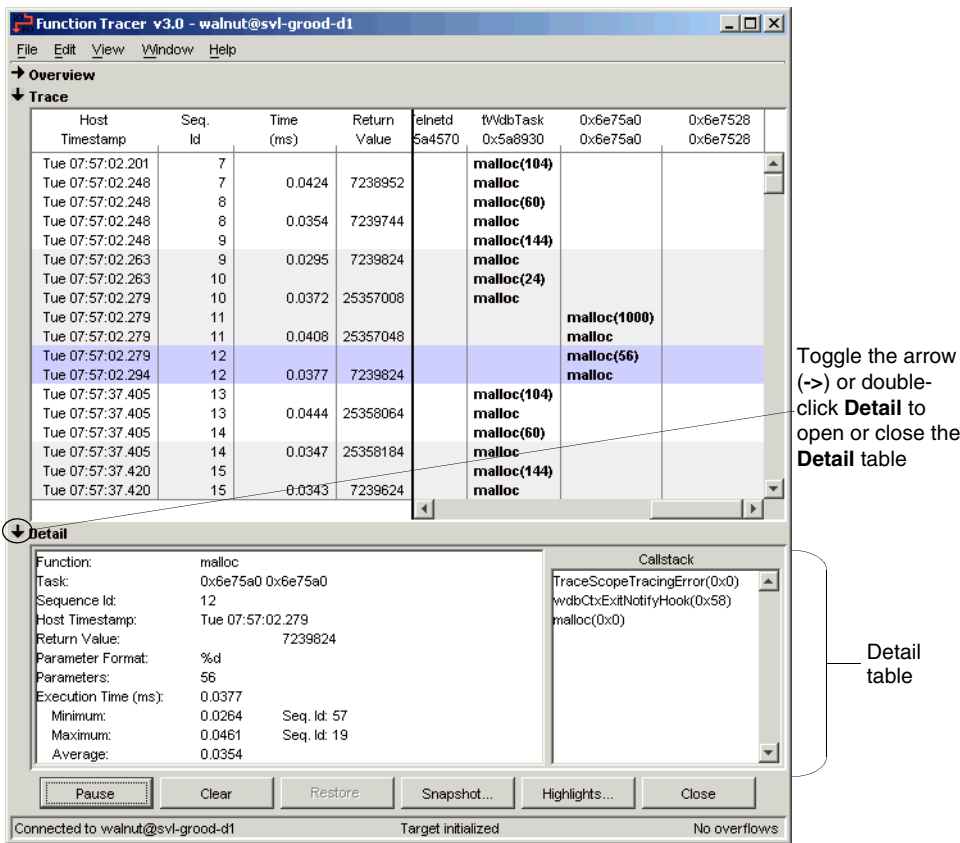
---

**NOTE:** When you resize the **Main** window, some column sizes may be readjusted to find a better fit for the columns within the table.

---

### Detail Table

The **Detail** table is initially hidden, but it can be displayed by toggling the small arrow to the left of the name or by double-clicking the **Detail** name itself. It contains all the available trace information for a trace point pair selected in the **Trace** window.



### Detailed Information

The left panel of the **Detail** table shows pertinent information about the selected trace point.

The information items are:

#### Function

The name of the function being traced.

#### Task

The task name and ID in which the function resides.

#### Sequence ID

The record number assigned sequentially by Function Tracer.

### Host Timestamp

The date and time of the trace call, from your host computer.

### Return Value

The return value for the function, in integer format.

### Parameter Format

A list of one or more formats for passed parameters, as configured in the **Parameter Format** field of the **Registration** window. For a description of the formats, see [Registering Trace Points](#), p.51.

### Parameters

The actual formatted values (with text descriptions, if any) of parameters passed to the function.

### Execution Time (ms)

The elapsed execution time for this function call, calculated from the high resolution timestamp driver. For a description of the driver, see [2.3 Starting Function Tracer](#), p.9.

### Minimum

The value and corresponding Sequence ID of the minimum execution time.

### Maximum

The value and corresponding Sequence ID of the maximum execution time.

### Average

The statistical average execution time over all trace points.

## Call Stack

The **Callstack** list displays the call stack for the currently selected function-entry record inside the **Trace** table. The call stack informs you of the list of nested functions calls that led up to the traced function, even if the nested functions calls are not traced. The call stack provides useful information without having to register too many functions with Function Tracer.

The format of the call stack list is:

```
func1 (offset1)  
func2 (offset2)  
...  
funcN (offsetN)
```

where **funci** is a function name and **offseti** is the offset address into the calling function at which the call to the next function occurred. The first function in the call stack is the traced function.

To view the source code for a function simply select its name in the Callstack field. The source code is displayed in a separate window (see [3.2.3 Source Code View Window](#), p.33). If Function Tracer cannot find the source code for a function, a dialog box is displayed allowing you to enter a list of directories to search (see [Source Path Dialog Box](#), p.35).



---

**NOTE:** The number of entries in the call stack is limited by the initialization parameter (see [2.3 Starting Function Tracer](#), p.9). If the actual call stack is deeper than this limit, the first function is still the traced function, but the last function in the call stack is not the top-level caller in the actual call chain.

---

## Buttons

Buttons located near the bottom of the window provide more convenient access to the most frequently used menu bar items. The buttons correspond to menu items of the same name, described in [3.2.2 Main Window](#), p.28.

The buttons are:

### Pause/Resume

Temporarily stops or starts (toggles) the display only of trace data being generated.

### Clear

Clears the **Trace** table.

### Restore

Restores all data removed with **Clear**.

### Snapshot

Copies the entire contents of the **Trace** table into a separate window.

### Highlights

Opens the **Highlight** window where you can select highlight criteria and colors.

### Close

Closes the window.

## Status Bar

The status bar appears at the bottom of both the **Main** window and the **Registration** window. Messages indicate the current status of the connection with the target.

The possible status messages fall into three categories:

1. Connection Status (left end), which can be

### Awaiting connection

The GUI is attempting to contact the target server for the target. The target-server name is indicated in the Title Bar of the window.

### Connected to *targetServer*

When the GUI attaches to the target server for the target, it displays this message where *targetServer* is the name of the target server.

### *target* daemon not responding

When the GUI was connected to the target server for *target*, but loses connection, it displays this message. Possible causes are:

- Target was rebooted.
- Network connection interrupted or disconnected.
- Target server is busy, or the machine running the target server is busy.

2. Target-Initialization Status (center), which can be

### Target not initialized

If the GUI is already connected to the target server, this message indicates that the target libraries have not been initialized yet.

### Target initialized

This indicates that the GUI is connected to the target server and the target is initialized successfully.

3. Overflow Status (right end), which can be

### No overflows

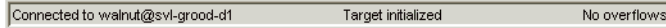
The message queue on the target that buffers trace records has not overflowed.

### Overflows *num*

The message queue on the target that buffers trace records has overflowed. The number, *num*, indicates the number of lost records. You may need to restart Function Tracer with a larger buffer, reduce the

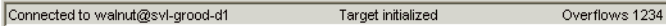
number of traced functions, or apply filtering to traced functions to reduce the number of records.

When Function Tracer is operating properly, the status bar should read:



Connected to walnut@svl-grood-d1      Target initialized      No overflows

If the target message queue overflows, the status bar will look like this example:



Connected to walnut@svl-grood-d1      Target initialized      Overflows 1234

## 4.4 Operational Features

Once Function Tracer has successfully launched and is displaying trace points in the Main window, there are additional features you can use to augment data display.

### Viewing Source Code

Any time while Function Tracer is running you can open and view the source code file for any function listed in the **Callstack** display in the **Details** view. To do this, select the desired function in the display and the Source Code View window opens displaying the source code file.

For details, see [3.2.3 Source Code View Window](#), p.33.

### Arranging Columns

Function Tracer displays a column in the output (**Main**) window for each task in your target program. If there are multiple tasks running on your target, you can choose which of the tasks to display in the window using the **Columns** dialog box. This reduces the horizontal scrolling required to see only the output you are interested in viewing.



For details, see [3.2.7 Columns Dialog Box](#), p.44.

### Setting Highlight Color

Output data in the output (**Main**) window is initially displayed in plain text (with no properties differentiating any criteria among the data). You can add color to provide this differentiation, using the **Highlight** window. The background of a data cell, or the entire line, can be colored according to a variety of criteria you select in this window. Examples of highlighted data are shown in Sections and above.

For details, see [3.2.6 Highlight Window](#), p.40.

### Taking Snapshots

The **Snapshot** window displays a static view of all the trace records received and stored since your target program was started. These records consist of all the entries displayed up to the instant the snapshot was taken. A snapshot is denoted by the word **Snapshot**, along with the filename, date and time of the snapshot, displayed in the title bar.

Use the **Edit > Snapshot** menu command to take a snapshot at any time. Multiple snapshots can be open at same time for comparison and analysis.

For details, see [3.2.4 Snapshot Window](#), p.37.

### Adding Custom Modules

Use the **Custom Modules** dialog box to add new modules and routines that were not loaded with the target libraries, or are not otherwise known to the target server.

For details, see [3.2.8 Custom Modules Dialog Box](#), p.45.

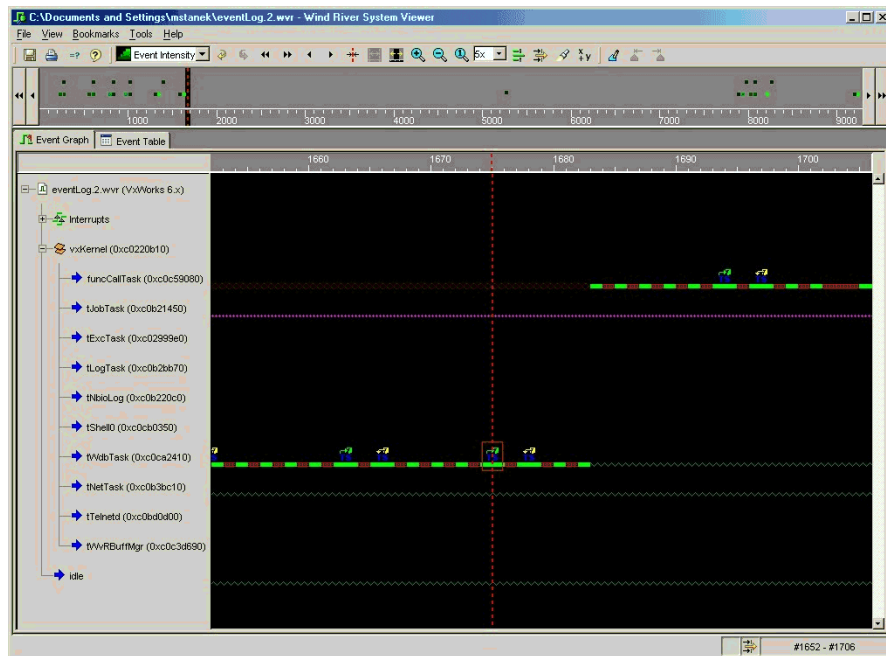
### Viewing Messages in the Console Window

The **Console** window is where Function Tracer reports errors and warnings. Its use is affected by the **Message Log Verbosity** level set in the **Function Tracer Launcher** dialog box (see [Starting Function Tracer From Workbench](#), p.11).

For details, see [3.2.5 Console Window](#), p.39.

## 4.5 System Viewer Event Integration

Function Tracer is integrated with the Wind River **System Viewer** to allow you to view trace events directly in a **System Viewer** window.



You can see the traced function-entry and function-exit events alongside other operating-system events, such as task switches and semaphore calls.

For details on installing, initializing and running the System Viewer, see the *Wind River System Viewer User's Guide*.

## Automatic System Viewer Support

When you load Function Tracer automatically from Workbench (see [2.3 Starting Function Tracer](#), p.9), the Function Tracer Setup code detects whether your target kernel supports System Viewer. If so, the Function Tracer target libraries are initialized with System Viewer support. With this support enabled, Function Tracer will post a System Viewer event for every trace record. Function-entry and function-exit events appear as two different types of events.

## High Resolution Time-Stamp Driver

When you load Function Tracer automatically from Workbench (see [2.3 Starting Function Tracer](#), p.9), the Function Tracer setup code detects whether your target kernel includes the high resolution time-stamp driver. If so, the Function Tracer target libraries are initialized with timestamp support that enables Function Tracer to record precision timing information for each traced function. The timing information is displayed with the function-exit trace records.



---

**NOTE:** Although the high resolution timestamp driver is described only in the *Wind River System Viewer User's Guide*, you can enable it without enabling System Viewer support for your kernel.

---

## System Viewer Events

The System Viewer events posted by Function Tracer have the following properties:

- **Function-entry**

Uses the System Viewer event ID, User0003. The properties of the event include the function name and address, as well as the sequence ID number of the corresponding trace record.

- **Function-exit**

Uses the System Viewer event ID, User0004. The properties of the event include the function name and address, as well as the sequence ID number of the corresponding trace record.

Use the sequence numbers in the System Viewer events and in the Function Tracer **Trace** table to match events with trace records.



# 5

## *Usage Tips*

- 5.1 Introduction 71
- 5.2 Observing Practical Limitations 72
- 5.3 Tracing Tips 76

### 5.1 Introduction

This chapter provides practical knowledge for proper use of Wind River Function Tracer. Read this chapter thoroughly **before** using Function Tracer on anything other than the example given in [2.4 Testing Your Installation](#), p.14.

The practical limitations on tracing routines you are likely to encounter include the effects of missing symbols, routines that must never be traced, and the practical limits on the number of routines being traced simultaneously. The best methods to use when tracing routines called very frequently are outlined as well.

Finally, some special limitations imposed by Function Tracer when tracing routines that return floating point values are explained.

## 5.2 Observing Practical Limitations

The conditions and events outlined in this section need to be taken into serious consideration when configuring Function Tracer to run on your project. Disregarding the warnings given here can lead to wasted time and effort due to incomprehensible errors, and even system crashes. You are urged to read and heed these tips and avoid the difficulties.

### Missing Symbols

Function Tracer is designed to work in the absence of symbols. This can yield stack traces on the host that have only hexadecimal address values when a matching symbol is not found. This is normal.

### Processor Load and Bandwidth Considerations

Function Tracer is a software-based tool, and therefore requires processor time and resources to work. While the tracing process is designed to be as efficient as possible, there is an impact on other tasks running on the processor. That impact scales with the following events:

- number of trace points used (maximum allowable: **32**)
- maximum stack depth selected (maximum allowable: **64**)
- information selected to be returned for each trace point
- total number of tasks running
- processor throughput (clock frequency, memory bandwidth, and so forth)
- speed of the link to the host

Of all these parameters, only the first three are usually configurable. The number of trace points should be set to no more than two or three on slow or moderately loaded systems. The maximum stack depth should be set to no more than necessary to find the calling sequence of interest. Selecting more stack depth than necessary slows the system considerably.

Remember that the first two entries on every stack are obtained free, that is, they are obtained with no need to walk the stack and inspect program text regions. On some heavily loaded systems, or while tracing a very frequently called routine, one trace point at a level of two is often the only viable configuration.

Routines That Must Not Be Traced

There are routines which cannot be traced without causing a task to fail or the target system to crash. In general, no interrupt service routines (ISRs) can be traced and no system calls made directly or indirectly by Function Tracer during the critical analysis phases for patched routine entry and exit. A list of directly called routines in the critical path is given in [Table 5-1](#).

Table 5-1 Routines Directly Called by Function Tracer Along its Critical Path

Routine	Parameters
aim*( )	(aimMmuContextTbl( ), and so forth)
cache*( )	(cacheLock( ), cacheInvalidate( ), and so forth)
call*( )	(call( ), callExcHandler( ), and so forth)
cfront_*( )	(cfront_demangle_name( ), and so forth)
check*( )	(checkTaskSwitch( ), and so forth)
commonExc*( )	(commonExcStubCode( ), and so forth)
emptyWorkQueue( )	(excIntStub( ), excJobAdd( ), and so forth)
exc*( )	
int*( )	(intLock( ), intUnlock( ), and so forth)
isrDispatcher( )	
isrIdSelf( )	
job*( )	(jobAdd( ), jobQueuePost( ), and so forth)
kernel*( )	(logLevelChange( ))
logMsg( )	
lstAdd( )	
lstDelete( )	
lstLast( )	
moduleSegFirst( )	

Table 5-1 **Routines Directly Called by Function Tracer Along its Critical Path** (cont'd)

Routine	Parameters
<b>objVerify( )</b>	
<b>qFifo*( )</b>	(qFifoGet( ), qFifoPut( ), and so forth)
<b>semGiveDefer( )</b>	
<b>sigWind*( )</b>	(sigWindKill( ), and so forth)
<b>strcmp( )</b>	
<b>symFindByNameAndType( )</b>	
<b>sysClkRateGet( )</b>	
<b>sysTimestamp*( )</b>	(sysTimestamp( ), sysTimestampFreq( ), and so forth)
<b>syscall*( )</b>	(syscallDispatch( ), and so forth)
<b>task*( )</b>	(taskMemCtxSwitch( ), taskLock( ), and so forth)
<b>tickGet( )</b>	
<b>vm*( )</b>	(vmStateSet( ), vmPageLock( ), and so forth)
<b>vxMem*( )</b>	(vxMemProbe( ), and so forth)
<b>wdb*( )</b>	(wdbCtxSuspend( ), wdbExcEventGet( ), and so forth)
<b>wind*( )</b>	
<b>workQ*( )</b>	(workQDoWork( ), workQueueEmpty( ), and so forth)

The list of indirect calls is almost impossible to generate, being dependent on architecture and system features selected. For example, if a timestamp service is available, there is usually an architecture-dependent API that directly accesses the timer device. You should be aware of that API and not attempt to trace it. The same applies to many kernel services such as the scheduler and interrupt/exception handling routines. All the routines listed in [Table 5-1](#) have indirect calls associated with them in the kernel or the BSP code.



The kernel, in general, and the Run-Time Analysis Tools support library modules should not be traced. Attempting to do so will have destructive results, putting Function Tracer on an endlessly recursive analysis path through its own code, or causing the processor to crash due to nested exceptions.

## Tracing Frequently Called Routines

5

Routines that are heavily used, such as **semGive()** and **semTake()**, require that special precautions be taken before activating a patch. You must configure Function Tracer to run as fast as possible by following these steps:

1. If you have Function Tracer connected, disconnect it.
2. Start Function Tracer by connecting it to the target, but with a maximum stack depth of 2.
3. In the **Registration** window, open a module and select a symbol, or type the name of a symbol in the **Name** field.
4. Select only that one symbol in the **Function** list to highlight it.



**CAUTION:** Do not select the patch activation box or the **Activate Tracing** button!

5. Clear the Parameter Format string and check **Hide** for **Return Value**.
6. For **Filtering**, select **Display calls from selected tasks only**, then select the box to the left of the task of interest in the task list window.
7. Click **Apply Changes**.
8. Click **Activate Tracing** to begin tracing. If large overflows are reported, immediately click the button again to stop tracing.

These steps ensure the fastest analysis possible for Function Tracer and should enable you to trace heavily used routines in most cases. In some extreme cases the processor is too slow to handle the extra load of Function Tracer together with the heavily used routine, resulting in buffer overflow messages and even a task or system failure. Such extreme cases require a hardware tracing tool like a logic analyzer, an in-circuit emulator, or a JTAG tool.

If Function Tracer does keep up with the heavily used routine, you can now try adding input parameters and/or a return value. If that also works, you can then experiment with adding one or two to the maximum stack depth, but be advised that you could rapidly reach a point where buffer overflows are too severe, or the task or system begins to fail, or both.

The overall goal here is to minimize the analysis time and host link bandwidth so that a heavily used routine can be safely traced to as deep as possible in one task's stack.

## 5.3 Tracing Tips

### Tracing Routines Returning Floating Point Values

The current version of Function Tracer properly shows single precision floating point return values from software floating point routines. However, routines returning double precision values, or those utilizing a hardware floating point coprocessor are not rendered properly.

### Tracing Real-Time Processes

Often overlooked in downloading modules, including Real-Time Processes (RTPs), is the option to download local symbols as well as global. Tracing all modules, especially RTPs, is greatly enhanced by using this option. For more information on setting and using this option, see [6.2 Loading and Initializing Function Tracer Manually](#), p.78. Also see Cause #2 and Solution #2 in [Issues With the Target](#), p.82. Always be sure to use this option when you load Run-Time Analysis Tools libraries.

# 6

## *Troubleshooting*

- 6.1 Introduction 77
- 6.2 Loading and Initializing Function Tracer Manually 78
- 6.3 General Troubleshooting Tips 81

### 6.1 Introduction

This chapter helps you deal with the Wind River Function Tracer status and error messages.

If you get error messages, or are having problems getting Function Tracer to work, check the error messages in this chapter to see if they resolve your problems. If you are still unable to get Function Tracer to work, contact Wind River Technical Support.

## 6.2 Loading and Initializing Function Tracer Manually

If automatic launching fails, you must determine which libraries to load yourself, load them, then initialize Function Tracer manually. This section describes the process in detail.

Manual loading of Function Tracer is more involved than automatic loading. We strongly recommend you create a shell script that you source from the host shell, to invoke Function Tracer loading and initialization. This script contains lines that load the proper libraries and initialize Function Tracer.

### Load the Required Libraries

To load a library, type in a shell window, or add to the script file, a line using the following syntax:

```
-> ld 1 < WIND_SCOPETOOLS_BASE/target/arch/targetArch/library
```

where *WIND\_SCOPETOOLS\_BASE* (an environment variable of the same name) is the root of the tree where you installed Function Tracer, *targetArch* reflects the target processor version you are using, and *library* is the library to load.

The "1" (number one) flag in the **ld** command causes the local symbols to be loaded along with the global symbols; we recommend you always use this flag when debugging or using Wind River tools with your code.

[Table 6-1](#) lists the libraries needed by the Function Tracer target agent.

Table 6-1 Required Target Libraries

Target Configuration	Target Libraries
VxWorks 6.6	scopeutils.so
	patchutils.so
	tracescope.so
VxWorks 6.6 with RTP support	strtpsupportlib.so

### ScopeUtils and PatchUtils

The **ScopeUtils** and **PatchUtils** libraries contain the routines shared by the Function Tracer target agent and other tools in the Run-Time Analysis Tools

collection. You need to load the **scopeutils.o** and **patchutils.o** libraries onto your target.

```
-> ld 1 < WIND_SCOPETOOLS_BASE/target/arch/targetArch/scopeutils.so  
-> ld 1 < WIND_SCOPETOOLS_BASE/target/arch/targetArch/patchutils.so
```

### RTP Support

For systems with RTP support configured into them, this library must be loaded, in addition to those listed above, in order to enable tracing of the RTP tasks.

```
-> ld 1 < WIND_SCOPETOOLS_BASE/target/arch/targetArch/strtpsupportlib.so
```

### Function Tracer

Load the **tracescope.so** library for your architecture.

```
-> ld 1 < WIND_SCOPETOOLS_BASE/target/arch/targetArch/tracescope.so
```

### Initialize Function Tracer

To initialize the Function Tracer target agent, make the following calls in the target shell or in the target-shell script file:

```
-> stRtpSupportInit(32)  
-> TraceScopeInit(numMsg, verbosity, stackDepth, maxArgs, maxStrings,  
                  woEnable, tsEnable, triggerEnable, textStartAdrs, textEndAdrs)
```

The parameters are defined as follows:

#### *numMsg*

The size of the message queue on the target. The target must queue a message for every trace point it encounters. These messages are queued on the target until they are retrieved by the host. If this queue is too small, messages will be lost, and trace records will be missed; in this event, the target agent notifies the host GUI. To avoid missing records, you may need to adjust this parameter.

#### *verbosity*

Used to indicate the amount of debug messages printed by the Function Tracer target agent. These messages appear in the shell from which you run **TraceScopeInit()**. A value of 0 displays only error messages. Increase the value (in the range of 1-3) to display more debug messages.

*stackDepth*

The maximum depth of the call stack to record for each trace point. The call stack always starts from the lowest-level call, for example, the traced function. The maximum value for this parameter is **16**.

*maxArgs*

The maximum number of function parameters to record for each trace point. The maximum value for this parameter is **10**.



---

**NOTE:** A function parameter of type **double** counts as two parameters.

---

*maxStrings*

Maximum number of string buffers—each **80** characters in length—to store string arguments. String arguments of traced functions must be copied and saved until retrieved by the host. Use this parameter to specify the maximum number of strings to be queued. If this buffer overflows, the host GUI displays an overflow message in the lower right corner of the Function Tracer window.

*wvEnable*

Set to **1** for System Viewer support; otherwise set to **0**.

*tsEnable*

Set to **True** if your target supports the high resolution timestamp driver and you want Function Tracer to time the duration of traced functions.

*triggerEnable*

Set to **True** if your BSP has the triggering functions built in; otherwise set to **False**.

*textStartAdrs*

Use the value for the address of **sysInit( )** routine for VxWorks.

*textEndAdrs*

Use the value for the **frame\_info\_end** absolute symbol (look up this symbol in a host shell connected to the target).

## Example Target Script

The following is a complete example of a target-shell script to load and initialize Function Tracer on a target that supports TCP/IP:

```
ld 1 < WIND_SCOPETOOLS_BASE/target/arch/ppc603Vx6.6gcc4.1.2/  
scopeutils.so  
ld 1 < WIND_SCOPETOOLS_BASE/target/arch/ppc603Vx6.6gcc4.1.2/
```

```
patchutils.so
ld 1 < WIND_SCOPETOOLS_BASE/target/arch/ppc603Vx6.6gcc4.1.2/
      strtptsupportlib.so
ld 1 < WIND_SCOPETOOLS_BASE/target/arch/ppc603Vx6.6gcc4.1.2/
      tracescope.so
stRtpSupportInit(32)
TraceScopeInit(5000, 0, 5, 5, 100, 0, 0, 0, 0x001358)
```

### Starting Function Tracer Manually from the Command Line

Function Tracer is started from a batch file (.bat), with a fixed set of initialization parameters.



**CAUTION:** Before entering any other commands in the Host Shell, type:

```
run wrenv -p vxworks-6.6
```

This properly sets up the environment variables to allow you to start Function Tracer using the **tracescope** command described below

To start Function Tracer manually from the command line, use the **cd** command to change to the following directory:

```
WIND_SCOPETOOLS_BASE
```

where *WIND\_SCOPETOOLS\_BASE* (an environment variable of the same name) is the root of the tree where you installed Function Tracer.

The command-line syntax for using the *tracescope.bat* file to start Function Tracer is:

```
tracescope
```

If you have executed the instructions in [6.2 Loading and Initializing Function Tracer Manually](#), p.78, the Function Tracer GUI should open, ready for input.

## 6.3 General Troubleshooting Tips

This section organizes problem areas by the major components in which they occur.

## Issues With the Target

### ▪ Call Stack Display

Function Tracer does not appear to be displaying the proper call stacks for memory-allocation data points.

#### Cause #1

The target server was not started with the **-A** option.

#### Solution #1

You **must** start the target server with the **-A** option. This ensures that the target server loads local symbols in addition to global symbols. If this option is not selected, the call stack traces pick the nearest global symbol for calls from local symbols.

For more details, see [Target Considerations](#), p.10.

#### Cause #2

You did not manually load libraries with local symbols, so Function Tracer instead shows function names that are the nearest global symbols.

#### Solution #2

Make sure you load your libraries with local symbols as follows:

**1a 1 < ...**

In addition, when running RTPs, enable them using the following steps:

- In the **Project Explorer** view, right-click on the appropriate.vxe file.
- From the menu that opens, select **Run RTP on Target**.
- In the **Run** window that opens, click the **Edit** button on the **Advanced Options** line.
- In the **Advanced Options** window that opens, click the **Select** button on the **Options** line.
- In the **Options** window that opens, be sure the check boxes for the options **RTP\_GLOBAL\_SYMBOLS** and **RTP\_LOCAL\_SYMBOLS** are selected.
- Click **OK** to exit the Options window, then click **OK** again to exit the Advanced Options window, then click **Apply** to save your selections, and finally, click **Run** to run the RTPs on the target.

#### Cause #3 (for x86 targets only)

Make sure your target kernel was compiled with frame pointers enabled.



### Solution #3

Frame pointers enabled is the default with the compiler, but make sure you did not disable them when you last compiled. If you did, you must recompile your code with frame pointers enabled.

- **Target Connection Lost**

You may also receive the following message in the target shell:

```
Link ERROR: Broken Pipe
Error sending records, reconnecting...
```

If so, it means the target has replied back to the host, and the host has shut down the target. In this case, the error is possibly caused by the target not responding to the host within the specified target timeout period. You can adjust priorities and timeouts as follows, then retry.

Increase the priority of Function Tracer to a value just below the tWdbTask priority, and above the tNetTask priority. For instance, if tWdbTask priority is at 3 and tNetTask priority is at 50, set **Task Priority** in the **Connect to Target** dialog box to 9, reconnect, and try again.

You may also need to change the **Backend request timeout** value from the default 3 sec. to a higher number, such as 10. Do this (with your target disconnected) by right-clicking your target server in the **Remote Systems** view, then selecting **Properties** to open the **Target Connection** dialog box. In this dialog box, use the **Advanced target server actions** group in the **Target Server Actions** tab view to modify the timeout value as indicated above.

- **Target Crashes**

If a task, or the target system itself, crashes during tracing, you may not have a properly configured target operating system. In this case, you must rebuild the system.

When you rebuild the system, be sure to include the following components:

**INCLUDE\_MODULE\_MANAGER**

Supports module library calls.

**INCLUDE\_SHARED\_DATA**

Supports shared data regions when you are tracing VxWorks Real-Time Processes.

These components are required for safe operation of Function Tracer on VxWorks systems. Update your target system build configuration accordingly in the Workbench Project Explorer view, and rebuild the system project.

- **Target Kernel Start and End Addresses**

When DFW is unable to determine the target kernel text start or end address, the following warning is displayed:

```
Unable to locate the start and/or the end of kernel text address,  
which the tool needs in order to successfully connect to the target.  
Please enter the values manually below:
```

Enter the start and end addresses manually in the fields provided, and continue the startup process. However, if you do not know the exact layout of your target memory and cannot supply correct values, you must exit Function Tracer and rebuild your VIP project, adding to it the following symbols:

```
wrs_kernel_start_text  
wrs_kernel_end_text
```

This enables DFW to provide the needed addresses.

- **Degraded Performance when Running RTPs**

If your target code contains RTPs, and you start it running only after you have connected Function Tracer and started the GUI, you may experience an unacceptable level of slow response from Workbench and the target. This may be manifested in any of the following ways:

- The RTPs may take a very long time (up to several minutes) to become fully operational, and even longer for symbols to begin showing up in Function Tracer.
- A simple workaround for this is to start the target program running first, then connect and start the Function Tracer GUI. This works because the slow-loading RTPs are loaded, or nearly so, before Function Tracer starts and begins its memory-intensive communication activities over the target connection.
- You may also notice that some symbols are unresolved when the target code is first started. This is because the first calls into the new RTP's memory library are captured by the Function Tracer GUI before the RTP task ID and symbols have been registered by Workbench. The workaround described above also prevents this behavior.
- Under certain conditions you may experience an even greater lack of response. If your RTP spawn time limit is short (say 30 seconds or less), you may see the message,

```
Failed to launch RTP name.
```

If the spawn time limit is longer and the RTP actually launches, you may see the message,

Target OS object not found.

The workaround either of these conditions is to increase the priority of the RTP and try again.



**CAUTION:** If you are running an RTP on your target **that must be started before Function Tracer**, you must increase the RTP's initial task priority from the default 100 to a value of about **60** (higher than the target agent but lower than the network task) to enable the RTP to execute cooperatively with Function Tracer.

In addition, the RTP spawn time limit must be set to **120** seconds or greater, and the backend request time limit must be set to **30** seconds. With the target disconnected, edit these values in the **Advanced target server options** panel of the **Target Server Options** tab view in the target **Properties** dialog box.

If you do not attend to these items, the RTP initialization task may not receive sufficient CPU time to complete its execution before the RTP spawn time limit expires and causes the host to stop all tasks running in the RTP.

For more information, see *Wind River Workbench User's Guide, VxWorks Version: RTPs and Shared Libraries from Host to Target*.

#### ▪ **Frame Pointer Generation**

Both Gnu and Diab (Wind River) compilers normally generate required frame pointers. However, there are some compiler options that turn them off. These options *must* be avoided. They are:

For the **Gnu** compiler:

- **-fomit\_frame\_pointer**
- **-fomit\_leaf\_frame\_pointer**

For the **Diab** compiler:

- **-Xkill-opt=0x800000**

## **Issues With the Host**

When the GUI application detects errors, it displays a window containing error messages. This section describes some of the error messages you might encounter and the possible causes and solutions.

- **Target Server Restarted**

If the target server for your target was restarted without rebooting the target, you will see the message:

```
Target server restarted - please reload target libraries
```

In this case, you need to reboot your target. Otherwise, the newly started target server does not know about libraries already loaded onto your target. Function Tracer resets itself after you have loaded the target libraries.

- **Disconnected from Target Server**

If Function Tracer detects that its tool has become disconnected from the target server, you will see the error message:

```
wtxError: Not connected to target server, trying to reconnect.
```

Function Tracer will attempt reconnection after a short delay.

- **Target Rebooted**

Function Tracer detects target reboots and displays the message:

```
Target was rebooted, please ensure target libraries are reloaded.
```

Be sure to reload the target libraries by selecting the **Function Tracer Launch** icon in Workbench. Function Tracer attempts to initialize the target libraries again after a short delay.

- **Lost Connection**

If Function Tracer loses connection with the target, you will see the error message:

```
wtxError: Connection temporarily lost. (reboot?) Will try to  
reconnect in 10 seconds.
```

This could be caused by target crashes, disconnected network wires, or busy machines. Check your connections and your target. Function Tracer attempts reconnection after 10 seconds.

- **Out of Target Memory**

If you specify a message queue size that is too large for the available target memory, you will see the message:

```
Unable to allocate queue on target. Please press the Reload button  
and specify a smaller queue size.
```

Click **Reload** in Function Tracer and specify a smaller queue size.

- **Not Initialized**

If you are launching Function Tracer automatically (see [Starting Function Tracer From Workbench](#), p.11) and the target libraries have not been initialized, or if Function Tracer was unable to allocate memory for internal data structures, you may see one of the following messages:

```
Target not completely initialized. Please press the Reload button to
load the libraries and specify a smaller queue size.
```

or

```
Unable to determine stack depth on target....Please initialize the
target libraries.
```

Click **Reload** in Function Tracer and specify a smaller queue size. Function Tracer tries to initialize the target libraries again after a short delay.

- **Target not completely initialized**

This error occurs when you try to initialize the target with a queue size specified in the **Function Tracer Launch** window that is too large.

```
Target not completely initialized. Please reload the target libraries
and specify a smaller queue size.
```

To fix the error, select the Function Tracer icon in the Workbench toolbar, uncheck the **Start Function Tracer GUI on host** check box, and specify a smaller queue size.

- **Initialization**

You should never see the following initialization error messages:

```
Unable to determine rtiToolTGT address on target....Please initialize
the target libraries...
```

or

```
wtxError: WTX Handle is invalid, restarting the tool.
```

If you do see these messages, please contact Wind River Technical Support.

- **Registration**

The following message is displayed when you try to activate more than 50 functions:

```
Module module name contains count functions.
Continuing with activation of this many functions may
have a negative impact on the target's performance.
```

This is a warning message only; you may **Cancel** or **Continue** the request.

The following message warns that another Function Tracer may be attached:

There is another Function Tracer attached to the target server. Exit?

The message is displayed on startup if the target server detects that another Function Tracer may already be attached. If this is the case, then proceeding can cause loss of data by either or both Function Tracer applications. Unfortunately this situation can also occur when Function Tracer crashes and does not notify the target server. In this case, you may proceed without any loss of data. Click **Yes** or **No**.

The following message is displayed when you reboot the target:

Target server restarted, please reload target libraries.

You need to reload the libraries by selecting the Function Tracer icon in the Workbench toolbar and unchecking the check box labeled **Start Function Tracer GUI on the host**.

- **Custom Modules**

The following message is displayed when you click **Add Module** with no name in the **Module Name** field:

The module name cannot be blank!

There must be a name in the **Module Name** field before trying to add the module.

The following message is displayed when you click **Add Function** with no name in the **Function Name** field:

The function name cannot be blank!

There must be a name in the **Function Name** field before trying to add the function.

The following message warns you that you have entered a duplicate module name:

A module named "*name*" already exists!

The name you enter in the **Module Name** text field cannot match the name of a custom module that already exists.

The following message warns you that you have entered a duplicate function name:

Module "*module name*" already contains a function named "*function name*"!

When you click **Add Function**, the name in the **Function Name** text field cannot match the name of an existing function in the currently selected module.

- **Console**

The following message is displayed when there is a **Console** file save error:

*An error occurred while trying to save to file name*

This happens when you try to save the contents of the **Console** window to a file and an error occurs.

- **Export**

The following message is displayed when there is an **Export** file write error:

*An error occurred while trying to write to file name*

This message is displayed when you click **Export** to write trace data to a file and an error occurs while trying to write to the file.

- **Open Trace Data**

The following message is displayed when there is an error reading a trace data file:

*file name does not appear to be a valid trace data file!*

When you click **Open Trace Data** and select a file, Function Tracer scans the file, and if it does not recognize the data in it, displays the above error.

The following message can be displayed when trying to read a trace data file:

*Unable to read file name*

When you click **Open Trace Data** and select a file, this message is displayed if you do not have read privilege for that file.

The following message is displayed when trying to read a non-data file:

*file name is not a file!*

When you click **Open Trace Data** and select an item, this message is displayed if the item (usually a directory) is not actually a file.

- **Save Trace Data**

The following message is displayed when there is a **Console** file save error:

*An error occurred while writing to file name*

This is message is displayed when you click **Save** to write trace data to a file and an error occurs while trying to write to the file.

- **Print**

The following message is displayed when you have not supplied required parameters in the **Print** dialog box:

The "from" and "to" fields must contain valid integer values.

This message indicates that you selected the **By sequence id:** option but did not enter integer values into both the from and to text fields.

The following message is displayed when there is a printer error:

```
Print error "error from printer" .
```

This happens when an error occurs while you are trying to print trace data. The error contents are system and error specific.

- **License**

The **License Expired** window contains a very specific error message from the FLEXlm license manager. This message can be displayed at any time if either the FLEXlm license manager crashes or the runtime license expires.

The **License Checkout Failed** window, containing a very specific error message from the FLEXlm license manager, is displayed at startup if the application is unable to check out a license from the FLEXlm license manager.



# A

## API Reference

<b>NAME</b>	<b>TraceScopeInit( )</b> – track function calls on the target
<b>SYNOPSIS</b>	<pre>int TraceScopeInit( const int numMsg, const int verbosity,                     const int stackDepth, const int maxArgs,                     const int maxStringBufs, const RTIBool wvEnable,                     const RTIBool tsEnable, const RTIBool triggerEnable,                     const RTIBool unused, void * textEndAdrs)</pre>
<b>DESCRIPTION</b>	
<b>PARAMETERS</b>	<p>numMsg - The size of the message queue on the target.</p> <p>verbosity - Used to indicate the debug level.</p> <p>stackDepth - The maximum depth of the call stack in the allocation record</p> <p>maxArgs - The maximum number of arguments to process for function entry.</p> <p>maxStringBufs - Maximum number of string buffers to hold string args (each 80 characters).</p> <p>wvEnable - Set to RTI_TRUE if BSP has the function wxEvent built in.</p> <p>tsEnable - Set ot RTI_TRUE if BSP has the function sysTimestamp built in.</p> <p>triggerEnable - Set to RTI_TRUE if BSP has the triggering functions built in.</p> <p>unused - Always set to 0.</p> <p>textEndAdrs - Must be set to the last valid kernel text address.</p>
<b>RETURN VALUE</b>	0 for failure, 1 for success.
<b>NOTE</b>	Not all architectures support dynamic code patching.



# B

## Glossary

### **activate/deactivate**

Function Tracer can maintain a large list of registered trace points, but it only monitors and logs calls to *active* trace points. You can activate and deactivate any trace point dynamically, as your system executes.

### **call stack**

The list of nested function calls that lead up to a traced function. For each trace-log entry, Function Tracer includes the call stack for that call.

### **filter**

To prevent large amounts of log data, you can specify for each trace point a list of tasks to ignore or a list of tasks to watch.

### **graphical user interface (GUI)**

The collection of computer programs and the media-oriented screens, windows, dialog boxes, menus, and buttons they produce that provide for enhanced human-computer interactions with no, or minimal, keyboard input.

### **host**

The computer on which the Function Tracer GUI is running, which receives and processes the allocation record data collected from the target agent machine.

### **mangle/demangle**

C++ compilers encode the function name, class name, and parameter types for a function into a symbol name in a process known as *mangling*. Function Tracer has the ability to reverse this process (*demangle*) for function names it finds on the call stacks.

### **module**

A collection of functions, such as a library or executable.

### **offset**

In a call stack, this is the memory distance (in bytes) from the start of the current routine to where the call to the next function occurs.

### **parameter format**

As part of the registration of a function, you can supply a format string—like the format string to **printf()**—to specify how you want the function call and its parameters to appear in the graphical log.

### **patch**

The process of changing run-time code dynamically, without compilation. When you activate a trace point, Function Tracer patches the corresponding function to insert additional code that collects the function-entry and function-exit trace records.

### **poll rate**

The rate at which Function Tracer sends a request to the target agent for allocation messages.

### **register**

Registering a trace point with Function Tracer involves defining the name of the function to be traced, a parameter-format string that defines the number and data types of the function arguments, and a list of task filters. The registration information enables Function Tracer to print the log entries in easily readable, user-specified format. A trace point must be activated before Function Tracer can monitor its execution.

### **routine**

Used interchangeably with *function*.

**snapshot**

A copy of the trace log at an instant of time may be copied and saved to a **Snapshot** window. A snapshot makes it easier to compare data between different runs.

**target agent**

This refers to the part of Function Tracer that runs on the target.

**trace point**

A function that has been registered with Function Tracer for monitoring. A trace point also is referred to as a *traced function*.

**trace record**

Function Tracer creates a record when entering a traced function and another one when exiting the traced function. A function-entry record includes the function parameters and the call stack. A function-exit record can include the function-return value and timing information.

**verbosity**

Controls the type and number of messages generated by the target or GUI and displayed either in the target console or the GUI console.

**view source**

Displaying the source code for the selected module.



# Index

## Symbols

%c character [52](#)  
%d decimal int [52](#)  
%e float [52](#)  
%f float [52](#)  
%g float [52](#)  
%le double [52](#)  
%lf double [52](#)  
%lg double [52](#)  
%p pointer [53](#)  
%s string [53](#)  
%u unsigned int [53](#)  
%x hexadecimal [52](#)

## A

activating trace points [26](#), [54](#), [93](#)  
adding custom modules/functions [45](#)  
architecture  
    host [2](#)  
    message queue [3](#)  
    summary [2](#)  
    target [2](#)  
automatic System Viewer support [69](#)

## B

button bar  
    Main window [64](#)  
    Registration window [27](#)  
buttons  
    Activate/Deactivate Tracing [28](#)  
    Apply Changes [28](#)  
    Clear (Trace table) [64](#)  
    Clear Changes (trace point) [28](#)  
    Close [30](#), [41](#), [43](#), [47](#), [48](#), [64](#)  
    Highlights [28](#), [64](#)  
    Pause/Resume [64](#)  
    Restore [64](#)  
    Snapshot [64](#)

## C

C++  
    demangle [94](#)  
call stack [3](#), [93](#)  
    depth [64](#), [80](#)  
    format [63](#)  
    offset [94](#)  
clearing, trace log [31](#)  
colors, trace records [40](#)  
Columns dialog box  
    description [44](#)

- opening [32, 44](#)
- rearranging in tables [58](#)
- using [66](#)
- View menu command [32](#)
- compiler
  - options [8](#)
  - warnings [9, 85](#)
- connection status [65](#)
- Console window
  - description [39](#)
  - opening [33, 39](#)
  - title bar [40](#)
  - using [67](#)
  - verbosity debug messages sent [13](#)
  - View menu command [33](#)
- copying trace log [95](#)
- Custom Modules dialog box
  - closing [48](#)
  - description [45](#)
  - loading custom modules [47](#)
  - opening [32, 45](#)
  - return value [48](#)
  - screen elements [46](#)
  - using [67](#)
  - View menu command [32](#)

## D

- deactivating trace points [26, 55, 93](#)
- demangle [94](#)
- description [3](#)
- Detail table
  - call stack [63](#)
  - description [61](#)
  - displaying [61](#)
- dialog boxes
  - Columns [44](#)
  - Custom Modules [45](#)

## E

- Edit menu items

- Clear (Trace tables) [31](#)
- Snapshot [31](#)
- error messages [85–90](#)
  - Console [89](#)
  - Custom Modules [88](#)
  - Disconnected from Target Server [86](#)
  - Export [89](#)
  - Initialization [87](#)
  - License [90](#)
  - Lost Connection [86](#)
  - Not Initialized [87](#)
  - Open Trace Data [89](#)
  - Out of Target Memory [86](#)
  - Print [89](#)
  - Registration [87](#)
  - Save (Trace Data) [89](#)
  - Target not completely initialized [87](#)
  - Target Rebooted [86](#)
  - Target Server Restarted [86](#)
  - troubleshooting [85](#)
- examples
  - ignoring tasks [18](#)
  - malloc() [14](#)
  - message queue [16, 21](#)
  - Registration window [17](#)
  - sequence ID [19](#)
- exporting trace data files [30](#)

## F

- File menu items
  - Close [30](#)
  - Export [30](#)
  - Open Trace Data [30](#)
  - Print [31](#)
  - Quit [31](#)
  - Save [30](#)
- File Search dialog box [36](#)
- files, required, table of [78](#)
- filtering
  - ignoring tasks, description [27](#)
  - ignoring tasks, in example case [18](#)
  - ignoring tasks, registering trace points [53](#)
  - trace point task, definition of [93](#)



- trace point task, registering 53
  - watching tasks, in registering trace points 53
  - watching tasks, in registration 27
- function
  - list 25
  - name 60
  - parameters 3, 60
  - routine 94
- Function Tracer, list of windows 23
- function-entry record
  - in main window description 60
  - in trace record 95
  - in trace record description 3
  - name 60
  - parameters 60
- function-exit record
  - in main window description 60
  - in trace record 95
  - in trace record description 3
  - name 60
  - return value 59
- functions, list of 25

## G

- graphical user interface (GUI)
  - defined 93
  - host (Function Tracer) 2, 4

## H

- high resolution timestamp
  - description of 69
  - driver 69
  - Trace table data 59
  - unresolved symbols 13
  - used for elapsed execution time 63
  - used in initializing target agent 80
- Highlight window
  - closing 43
  - criteria table 41
  - description 40

- menu bar 40, 41
  - opening 32, 41
  - parameters 41
  - screen elements 41
  - title bar 41
  - using 67
  - View menu command 32
- highlights, trace records 40
- host
  - defined 93
  - GUI 2, 4
  - timestamp 59, 63

## I

- installation
  - testing 14

## J

- Java
  - JDK 8
  - JRE 8

## L

- Launcher dialog
  - starting Function Tracer automatically 11
- loading custom modules 47

## M

- Main window
  - button bar 64
  - description 56
  - Detail table 61
  - Edit menu 31
  - File menu 30
  - menu bar 30

- Overview table 56
- status bar 65
- task columns 60
- title bar 29
- Trace table 58
- View menu 32, 33
- Window menu 33
- menu bar
  - Edit commands 31
  - File commands 30
  - Highlight window 40, 41
  - Main window 30
  - Registration window 25
  - View commands 32
  - Window commands 33
- message queue
  - architecture 3
  - example 16, 21
  - size 65, 79
- modifying trace points 55
- modules
  - defined 94
  - list 25
- msgQReceive() 18

## N

- name
  - function 60
  - trace point 26

## O

- offsets
  - defined 94
- opening
  - Columns dialog 32, 44
  - Console window 33, 39
  - Custom Modules dialog box 32, 45
  - Highlight window 32, 41
  - Main window 33
  - Registration window 32, 33

- trace data files 30
- options
  - A, troubleshooting 82
  - command line 78
  - compiler warnings 9, 85
  - demangle 94
  - Launcher dialog box 11
- overflow status 65
- overview
  - real-time systems 1
- Overview table
  - description 56
  - displaying 56
  - rearranging columns 58

## P

- parameters
  - formats supported 52, 63
  - function 3
  - highlight criteria 41
  - maximum 80
  - registration 27
  - trace point 27, 94
- patch 2
- patch.so 78
- patchutils.so 78
- pausing, trace display 64
- poll rate
  - defined 94
  - setting, in Launcher dialog box 12
- printing trace data 31

## Q

- quitting Function Tracer 31

## R

- real-time
  - process components 8

- systems 1
- reboot
  - automatic reconnection 13
- recompile 4
- record
  - call stack 3, 63, 93
  - calling task 3
  - clearing 31
  - function name 60
  - function parameters 60
  - function-entry 60
  - function-exit 59, 60
  - high resolution timestamp 59, 63
  - host timestamp 59
  - sequence ID 3, 19, 59
  - timing 59, 63
- registering trace points 51, 94
- Registration window
  - button bar 27
  - description 24
  - function list 25
  - in example 17
  - menu bar 25
  - opening 32
  - parameters area 27
  - status bar 28
  - testing installation 15
  - title bar 25
  - View menu command 33
- removing trace points 56
- requirements
  - target 8
- resuming, trace display 64
- return value
  - Custom Modules dialog box 48
  - in trace record 3
  - trace point 27
  - trace table column description 59
- routine
  - see also* function
  - defined 94
- RTP
  - degraded performance 84
  - libraries needed 79, 82
  - support 8

## S

- saving trace data files 30
- scopeutils.so 78
- screen elements
  - Columns dialog box 44
  - Custom Modules dialog box 46
  - Detail table 61
  - Highlight window 41
  - Overview table 57
  - Registration window 25
  - Snapshot window 38, 67
  - Trace table 59
- scripts
  - initialize Function Tracer on TCP/IP target 80
  - initialize target agent 79
  - target shell 78, 80
- searching for source code files 35, 36
- selecting columns 44
- sendMessage() 17
- sequence ID 3, 19, 59
- shared data region support 8
- simulator, cannot be used as a target 10, 12
- Snapshot window
  - see also* Main window
  - description 37
  - edit command 31
  - snapshot, defined 95
  - snapshot, feature list 5
  - title bar 38
  - usage 67
- Source Code View window
  - described 33
  - using 66
- Source Path dialog box
  - alternate directory selection 35
- status
  - bar 28, 65
  - connection 65
  - overflow 65
  - target initialization 65
- string buffers
  - maximum 80
- strtpsupportlib.so 78
- System Viewer

- automatic support 69
- Function Tracer events 69
- initializing Function Tracer 80
- integration with Function Tracer 68

## T

### target

- agent architecture 2
- cannot be a simulator 10, 12
- initialization status 65
- server 8, 13, 14, 33
- shell script 78, 80

### target library

- example target script loading 80
- GUI initialization 13
- launching Function Tracer manually 78
- patch.so 78
- patchutils.so 78
- scopeutils.so 78
- strtpsupportlib.so 78
- table of required files 78
- tracescope.so 78, 79

### targetArch 78

### task

- bringing to top 33
- columns 60
- ID 3

### task filtering

- ignore 27, 53
- options 53
- trace point 5, 27, 93
- watch 27

### tExcTask 18

### timestamp

- high resolution 59, 63, 69, 80
- high resolution, driver 69
- high resolution, System Viewer 69
- host 59, 63
- unresolved symbols 13

### title bar

- Console window 40
- Highlight window 41
- Main window 29

- Registration window 25

- Snapshot window 38

### tNetTask 18

### trace data files

- exporting 30
- opening 30
- printing 31
- saving 30

### trace display

- clearing tables 64
- closing the window 64
- generating snapshot 64
- pausing 64
- restoring 64
- resuming 64

### trace log, clearing 31

### trace point

- activating 28, 54, 93
- clearing changes 28
- deactivating 28, 55, 93
- defined 51
- filtering 93
- highlighting 28, 40, 64
- managing 50
- modifying 28, 55
- name 26
- parameter format 27
- parameters 94
- registering 51, 94
- removing 56
- return value 27
- task filtering 5, 27, 53
- timing 69
- traced function 95

### trace record

- defined 95
- function-entry record in 95
- function-exit 95
- highlight, colors 40
- return value 3

### Trace table

- description 58
- feature list 4
- screen elements 59

- tracescope.so 78, 79

troubleshooting  
    -A option [82](#)  
    error messages [85](#)  
tWdbTask [18](#)  
    dialog box [14](#)  
    Launcher [11](#)  
WTX [4](#)

## U

uninstalling  
    JDK [8](#)  
    JRE [8](#)  
unresolved symbols, cause of [13](#)

## V

verbosity  
    defined [95](#)  
    initializing Function Tracer [79](#)  
    Warning [13](#)  
View menu  
    Columns [32](#)  
    Console [33](#)  
    Custom Modules [32](#)  
    Highlights [32](#)  
    Registration [33](#)  
    targetServer [33](#)  
viewing errors and warnings [39, 67](#)  
viewing source code [33](#)

## W

warnings  
    compiler options [9, 85](#)  
    verbosity [13](#)  
windows  
    Console [39](#)  
    Highlight [40](#)  
    list [23](#)  
    Main [56](#)  
    Registration [24](#)  
    Snapshot [37](#)  
Workbench