

WIND RIVER

Wind River® Workbench Data Monitor

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.

Wind River may refer to third-party documentation by listing publications or providing links to third-party Web sites for informational purposes. Wind River accepts no responsibility for the information provided in such third-party documentation.

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

1	Introduction	1
1.1	Introduction	1
	Data Monitor Overview	1
1.2	Architectural Summary	2
	VxWorks Targets	3
	Linux Targets	4
	The Host GUI	5
	Host-target Communication	5
1.3	Features	6
2	Getting Started	9
2.1	Introduction	9
2.2	Requirements	10
2.3	Starting Data Monitor	10
	Initializing the Target Server (VxWorks Only)	11
	Starting Automatically	11
	Starting Manually	14
	Usage Notes	18
2.4	The Data Monitor GUI	19

	Plot Window	20
	Plot XY Window	22
	Dump Plot Window	24
	Monitor Window	25
	Auxiliary Data-Display Windows	25
	Common Window Elements	29
2.5	Testing Your Installation	30
	On a VxWorks Target	30
	On a Linux Target	35
	Viewing the Signals	35
	Exploring the Demo Capabilities	35
	Automatic Signal Management (VxWorks Only)	36
3	Data Monitor Features	39
3.1	Introduction	39
3.2	Toolbars	40
	Main Toolbar	40
	Plots Toolbar	41
	Plot Window Toolbar	41
3.3	File Menu Item	43
3.3.1	Connect to Target	45
3.3.2	Load Snapshot	45
3.3.3	Save Snapshot	45
3.3.4	Load Config	46
3.3.5	Save Config	46
3.3.6	Plots	48
3.3.7	Signal Manager	49
3.3.8	Triggering	50
3.3.9	XY Signals	51
3.3.10	Derived Signals	52
3.3.11	Trace Log Window	53

3.3.12	Preferences	53
3.3.13	Close Window	69
3.3.14	Exit Data Monitor	69
3.4	Menu Bar	69
	Plot Menu Item (Windows Hosts Only)	69
	View Menu Item	70
	Window Menu Item (Windows Hosts Only)	70
	Help	70
3.5	Pop-up Menus	71
	On-Grid	71
	On-Trace (Windows Hosts Only)	73
	Signals Tree	73
	Legend	74
3.6	Screen Operations	76
	Zooming	76
	Markers	76
	Annotations	77
	Panning	77
	On-grid Measurements	78
3.7	Status Bar	78
4	Using the Signal Manager	81
4.1	Introduction	81
4.2	Using the Signal Manager Window	81
	Working With Signal Trees	82
	Installing Signals	84
	Disconnecting the Target	84
5	Triggering	85
5.1	Introduction	85
5.2	Configuring a Trigger	85

	Triggering Dialog Box - Windows Host	86
	Triggering Dialog Box - UNIX Host	90
5.3	Setting a Trigger	93
	The Chain of Events	93
	Trying it Yourself	94
6	Derived Signals	97
6.1	Introduction	97
6.2	Creating Derived Signals	98
	Mathematical Operations	104
	Troubleshooting Derived Signals	105
7	The Plot Window	107
7.1	Introduction	107
7.2	Plot Window Tour	108
	Selecting Signals	109
	Popup Menu	110
	Screen Operations	110
	Toolbar	110
	Menu Bar	111
	Signals Bar	116
	Legend Window (UNIX Hosts Only)	122
	Strip Chart	122
7.3	Signal Properties Dialog Box	123
7.4	Axis Properties Dialog Box (Windows Hosts Only)	128
7.5	Displaying Events	130
	Events Collected as Signals	131
	Events Collected as Markers	131
	Events Collected as Messages	133
7.6	Setting New Plot Window Preferences	134

8	The Plot XY Window	139
8.1	Introduction	139
8.2	Creating XY Signal Pairs	140
	Creating a Signal Pair	140
	Deleting a Signal Pair	141
	Modifying a Signal Pair	141
8.3	Plot XY Window Tour	141
	Displaying Signal Parameters	143
	Popup Menu	144
	Screen Operations	144
	Toolbar	145
	Menu Bar	145
	Signals Bar	149
8.4	Signal Properties Dialog Box	154
8.5	Setting New Plot XY Window Preferences	160
9	The Dump Plot Window	163
9.1	Introduction	163
9.2	Dump Plot Window Tour	164
	Displaying Signal Parameters	165
	Toolbar	165
	Menu Bar	166
	Signals Bar	168
9.3	Setting New Dump Plot Window Preferences	171
10	The Monitor Window	173
10.1	Introduction	173
10.2	Monitor Window Tour	174
	Displaying Signal Parameters	175
	Toolbar	175

	Menu Bar	176
	Signals Bar	178
10.3	Writing Data to the Target	181
	Using Writeback	182
10.4	Setting New Monitor Window Preferences	182
11	Working with Snapshots	185
11.1	Introduction	185
11.2	Utilizing Snapshots	185
	Taking Snapshots	186
	The Snapshot Process	186
	Saving Snapshots	187
	Loading Snapshots	193
	Exporting Snapshots in MATLAB and MATRIXX	193
	Deleting Snapshots	196
12	Displaying Remote Kernel Metrics	199
12.1	Introduction	199
12.2	Building an RKM Monitor Program	199
	On a VxWorks Target	200
	On a Linux Target	203
	Running the RKM Monitor From the Command Line	204
12.3	Viewing RKMs with Data Monitor	205
	On a VxWorks Target	206
	On a Linux Target	207
12.4	Troubleshooting	208
	On a VxWorks Target	208
	On All Targets	209

13	Using a VxWorks Target	211
13.1	Introduction	211
13.2	ScopeProbe Requirements	212
13.3	VxWorks Targets	213
	Building	213
	Automatic Loading and Running	213
	Manual Loading and Running	217
	Starting the Data Monitor GUI Manually	220
13.4	Troubleshooting	220
14	Using a Linux Target	225
14.1	Introduction	225
14.2	Building Your Application	225
	Adding Include Files	226
	Instrumenting Target Code	227
	Adding Libraries	227
	Compiling Target Code	229
	Testing Your Application	230
15	Installing Signals	231
15.1	Introduction	231
15.2	Using the Signal Installation Dialog Box	232
	Data Monitor Signal Installation Dialog Box	233
15.3	Installing With the Data Monitor API	241
15.4	Code Instrumentation Alternative	241
15.5	Removing Individual Signals	241
15.6	Process Notes	242
	Variable Expressions vs. Signal Names	242

	Hierarchical Signal Names	243
	Classes and Structures	244
16	API Introduction	247
16.1	Introduction	247
16.2	Using the Data Monitor API	248
	Initializing the Target Server (VxWorks Only)	248
	Registering and Activating Signals	249
	Setting Sample Rate	255
	Sampling Signals	255
16.3	Understanding Overflows	258
	Overflow Behavior	258
	Avoiding Overflows	258
	Notes and Hints	258
16.4	Triggering and Sampling Functions	259
16.5	Data Monitor Events API	260
	Setting Up	260
	Using	261
	Signals vs. Events	262
16.6	scope.ini File (VxWorks Only)	263
A	API Reference: VxWorks	267
B	API Reference: Linux	295
C	Data Monitor Demo Program	323
C.1	Introduction	323
C.2	Source Code for VxWorks	324
	VxWorks vxdemo.c Program	324
	Makefile for vxdemo.c	332

C.3	Source Code for Linux	334
	Linux scopedemo.c Program	334
	Makefile for scopedemo.c	345
D	MATLAB and MATRIXX Examples	347
D.1	Introduction	347
D.2	MATLAB Example	347
D.3	MATRIXX Example	351
E	RKM Signal Definitions	357
E.1	Introduction	357
E.2	Signal Descriptions	357
	Examples	363
F	Glossary	365
	Index	373

1

Introduction

1.1	Introduction	1
1.2	Architectural Summary	2
1.3	Features	6

1.1 Introduction

This chapter introduces you to the Wind River Data Monitor, a real-time graphical data monitoring tool for VxWorks or Linux targets.

Data Monitor Overview

Anyone who has developed real-time systems knows that getting the code written and compiled is only the first step. You still have to make it work. Where is the noise coming from? How full is the buffer? When did that valve open? What are the best parameters? Why did it do that? Understanding the system is the real challenge.

Data Monitor is a real-time graphical monitoring and data-collection tool that lets you monitor and analyze the dynamically changing values of variables in your real-time application while it is running. It is a powerful debugging aid for both hardware and software. With its multi-window environment, you can track down

performance problems, glitches, and program errors. Data Monitor presents a live analysis of the variables in your program while preserving real-time performance. You can immediately see the effects of code changes, parameter changes, or external events.

1.2 Architectural Summary

The Data Monitor architecture comprises two main components—a collection agent that runs on a real-time VxWorks or Linux target, and a GUI that runs in a Windows or UNIX host. Along with the GUI, two additional modules are used by the Workbench **Remote Systems Explorer** on both VxWorks and Linux targets:

- The DFW server to process symbol information sent by the data collection module, loaded on the target with your kernel (or RTPs for VxWorks).
- A Signal installation package for automatically or manually installing signals to be collected from your target and analyzed.

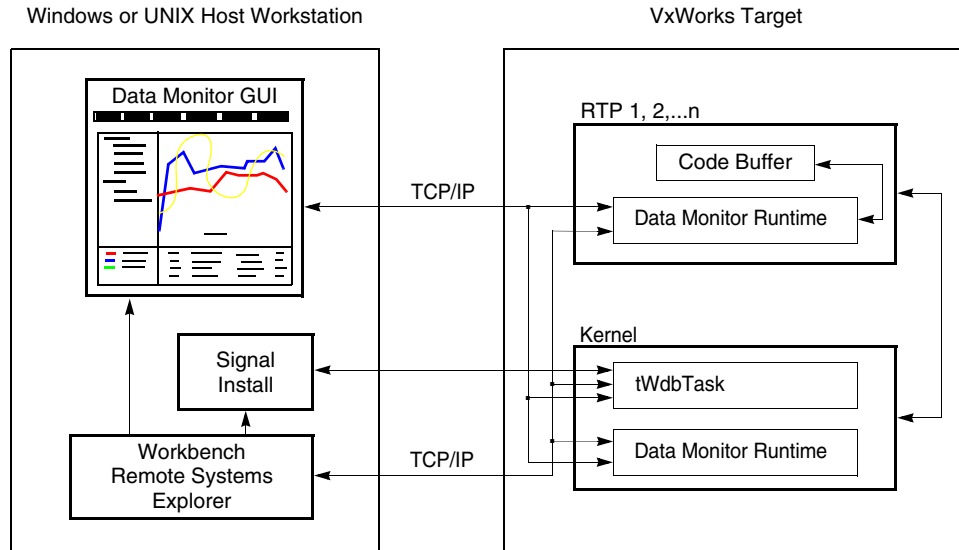
These components communicate through the Workbench Remote Systems Explorer, including the DFW server, over a TCP/IP link (or, for VxWorks only, the optional WTX link).

There are some minor differences in the implementation of target architecture between VxWorks and Linux targets. These differences are outlined in the following sections.

VxWorks Targets

1

The component architecture of Data Monitor running on a VxWorks target is as shown in the following illustration.



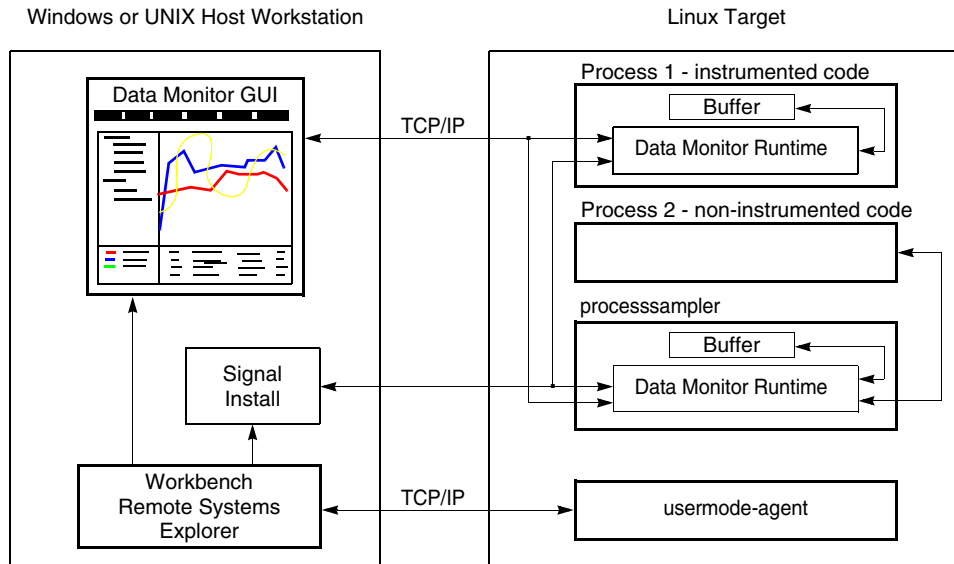
The components communicate through the VxWorks **Remote Systems Explorer**, including the DFW server, over a TCP/IP or optional WTX link

The Data Monitor for VxWorks application consists of the following modules:

- A multi-window graphical user interface (GUI) that runs on the host PC, providing dynamic views of your run-time data.
- A real-time data collection module loaded onto the target processor with your kernel or RTP(s). It collects and buffers time histories of variables in your program before sending them to the host for display.

Linux Targets

The component architecture of Data Monitor running on a Linux target is as shown in the following illustration.



The components communicate through the Linux **Remote Systems Explorer**, including the DFW server, over a TCP/IP link

The Data Monitor for Linux application consists of the following modules:

- A multi-window graphical user interface (GUI) that runs on the host (PC or workstation), providing dynamic views of your run-time data.
- A real-time data collection module that is loaded onto the target with your processes. It collects and buffers time histories of variables in your program before sending them to the host for display.

The Host GUI

1

In a Windows or UNIX host, the Data Monitor GUI allows you to view the data interactively as it is received. You can save the data on disk for later off-line analysis. Data can also be exported in a variety of formats.



NOTE: There are a few differences between the Windows and UNIX versions of the Data Monitor host GUI. Some are merely cosmetic, or only slightly different in wording, in which case only the Windows screenshots and wording will be used in this manual. However, where there are greater differences in the GUI layout, or in the selection of parameters presented, the differing aspects will be illustrated and described separately to help eliminate second-guessing and ambiguity.

Host-target Communication

The application task, running on the target module, is typically a user program, but it also can be a system process, especially with a real-time operating system. This task collects the data.



NOTE: Data Monitor is specifically designed to analyze only tasks written in C, C++, and **assembly** code; other languages will not work.

For VxWorks targets, Data Monitor supports both the TCP/IP and WTX modes of data transfer. For Linux targets, however, only the TCP/IP mode is supported.

In both of these modes, two additional low-priority tasks (threads), are in charge of transferring data to the host:

ProbeDaemon

The **ProbeDaemon** receives and processes commands from the host user interface. Through the ProbeDaemon interface, the target application can:

- Install variables to monitor.
- Change sample rates.
- Set triggers.
- Collect data.

LinkDaemon

This task transfers data to the host. During program execution, data is collected and placed in a local buffer. This data collection is very fast and is the only action that takes place at high priority (that is, at the priority of your application or an asynchronous sampling task). The LinkDaemon task,

running at very low priority, then takes the data from the buffer and sends them to the host.

The LinkDaemon stores the list of signals and the collected data samples on the target. These are sent to the host to be displayed at a later time. For details, see [16. API Introduction](#).

This mechanism is designed for minimal impact on real-time system performance.

1.3 Features

- **Real-time Graphical Display**

The Data Monitor full-color, real-time graphical displays let you watch your program execute. Multiple windows can be open at the same time, displaying a rich mixture of signals and functionality.

- **Minimal Intrusion**

Data Monitor does not impact the performance of your real-time system. Collection is very fast; data transfer takes place in the background at low priority.

- **Modify Data**

Data Monitor can also modify program variables. Experiment quickly, isolate problems, and run test cases by changing the value of variables and parameters while your program executes.

- **Dynamic Signal Installation**

Install variables by name as your program runs, including **structs**, **classes**, and **unions**, by simply typing in the name of the variable you want to view.

- **Data Storage**

Data Monitor exports data in many formats. It is organized (each run is timestamped and labeled with signal names and units), and accompanied by your notes. You can choose which data to save, or have Data Monitor save them automatically.

- **Support for Large Systems**

With this program, you can register literally hundreds of variables for monitoring. You can collect any subset of the registered variables, and organize your variables with a powerful hierarchical tree browser.

- **Type Support**

Data Monitor supports many data types without loss of precision. This includes support for all common data types—from one-byte char to eight-byte double, support for pointers and structures to make it easier to monitor complex data structures, user-defined buffers, and support for hexadecimal data-display.

- **Multiple Target Connections**

You can connect to more than one target during a Data Monitor session. This allows you to simultaneously view the results from multiple targets as needed on a single screen for live comparison. A specific example is found in viewing remote kernel metrics (see [12. Displaying Remote Kernel Metrics](#)).

2

Getting Started

2.1	Introduction	9
2.2	Requirements	10
2.3	Starting Data Monitor	10
2.4	The Data Monitor GUI	19
2.5	Testing Your Installation	30

2.1 Introduction

This chapter describes the main features of the Wind River Data Monitor graphical user interface (GUI). It presents a brief overview of the four distinctive data-display windows (**Plot**, **Plot XY**, **Dump Plot**, and **Monitor**), describing their many unique features and commands, as well as those they have in common. Each of these data-display windows is described in later chapters. We suggest you skim this section then run the demo program outlined in [2.5 Testing Your Installation](#), p.30.

2.2 Requirements

You must connect to your target in the Workbench **Remote Systems Explorer** view in order to use Data Monitor. For documentation on the Remote Systems Explorer view, consult the *Wind River Workbench User's Guide: Connecting to Targets*, as well as your platform User's Guide.

There are some dependencies Data Monitor places on your host operating system for resources that are specific to the target platform, summarized in the following section.

VxWorks Targets Only

- Data Monitor requires the use of the WDB agent. 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
- 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 more information on VIPs, see *Wind River Workbench User's Guide: VxWorks Image Projects*.

2.3 Starting Data Monitor

This section describes how to start Data Monitor running in a real environment. ([2.5 Testing Your Installation](#), p.30 shows you how to run the demonstration program).

Initializing the Target Server (VxWorks Only)

To run Data Monitor on a VxWorks target, you must first initialize your target server. You then need to load the *daemon* libraries and initialize the target by a call to **ScopeInitServer()**. This is followed by installing the signals you want to watch. A complete description of this process is given in [13. Using a VxWorks Target](#).

For both VxWorks and Linux targets, Data Monitor can be started in either of two ways:

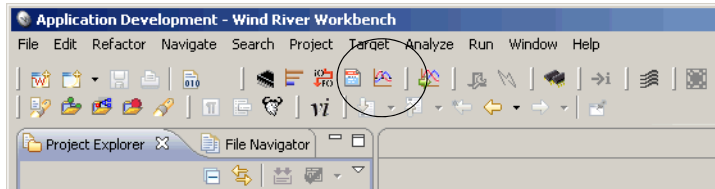
1. Automatically by clicking the **Data Monitor** button on the Workbench toolbar.
2. Manually on the host from a command line window.

Starting Automatically

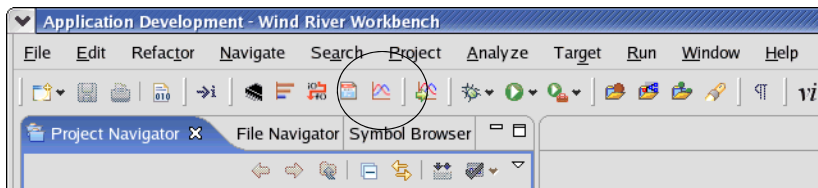
To launch Data Monitor from the Workbench toolbar and connect to your VxWorks target server (or Linux target), do the following:

1. In the **Remote Systems** view, connect to your target.
2. From your host GUI, select the **Data Monitor** toolbar icon (circled).

Windows Host



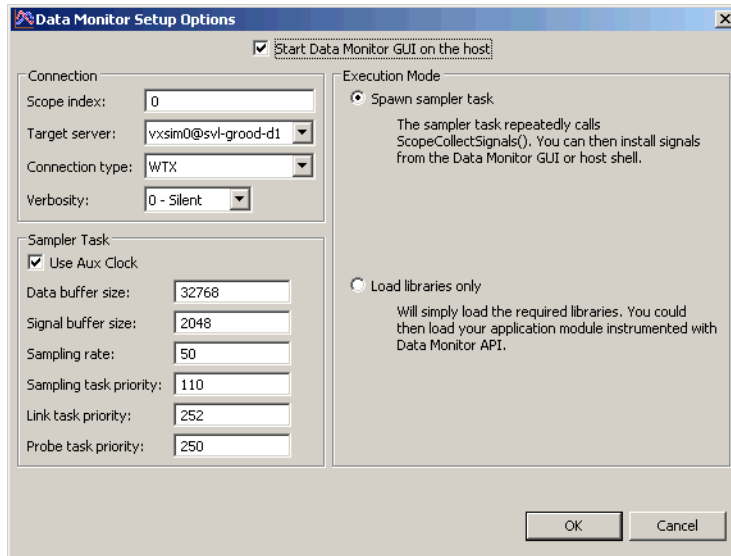
UNIX Host



The **Data Monitor Setup Options** dialog box opens where you can specify startup parameters for Data Monitor. This dialog box has significant differences depending on whether your target is **VxWorks** or **Linux**.

VxWorks Data Monitor Setup Options Dialog Box

For a VxWorks target, this **Data Monitor Setup Options** dialog box opens.



In this dialog box, select values for the following parameters:

- If Data Monitor is already running and you do not want to restart it (but only load libraries), deselect **Start Data Monitor GUI on the host** button and select **Load libraries only**. Otherwise, leave **Spawn sampler task** selected.
- Use the default value for the **Scope index** (or enter a value from **0-127** if you know a different value is needed). For more information on how this parameter is used, see [Scope Index](#), p.249.
- Select a **Target server** from the drop-down list of discovered target servers. If the list is empty, you do not have a target server running and you must create one first. For details on how to configure and start a target server, see the *Wind River Workbench User's Guide*.
- For **Connection type**, use the default **TCP/IP**. Using the default **TCP/IP** whenever possible is desirable because it is faster, but if you have special or unusual connection constraints, you can

choose the **WTX** connection type. Note that specifying WTX protocol when running WTX over a serial line can severely limit data throughput.



NOTE: For VxWorks, if you instrument an RTP with the Data Monitor API, you must choose the **TCP/IP** connection type when creating a connection to any Scope Index published by that RTP.

e. **Verbosity** has the following options to choose from:

0 (silent) - Displays only warning and error messages (most restrictive)

1 - Displays warning, error, and workflow messages.

2 - Displays warning, error, and greater volume of workflow messages.

3 (verbose) - Displays all system messages (most verbose).



CAUTION: Setting target verbosity to a value greater than 0 may cause the Usermode-Agent in the target (see [1.2 Architectural Summary](#), p.2) to needlessly generate a large number of messages.

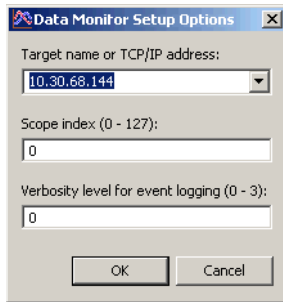
Generally, use the default value of 0 for target verbosity, unless requested by Wind River Technical Support to help you diagnose a problem.

- f. Choose **Spawn sampler task**, to perform asynchronous signal sampling, or **Load libraries only** to only load the target libraries.
- g. If you selected **Load libraries only**, make any desired modifications to the default sampling parameters. For a detailed description of the remaining parameter settings, see [Automatic Loading and Running](#), p.213.
- h. Click **OK** to accept your parameter selections and connect to your target.

For more details on launching Data Monitor for VxWorks, see [13. Using a VxWorks Target](#).

Linux Data Monitor Setup Options Dialog Box

For a Linux target, this **Data Monitor Setup Options** dialog box opens.



In this dialog box, select values for the following parameters:

- a. Select the **Target name or TCP/IP address** of your target from the drop-down list in the first text field, or enter it directly in the field.
- b. Use the default value for the **Data Monitor index** (or enter a value from **0-127** if you know the value needed). For more information on how this parameter is used, see [Scope Index](#), p.249.
- c. Use the default value of **0** for **Verbosity level for event logging** (or see the **Verbosity** options in Step "e" above).
- d. Click **OK** to accept your parameter selections and connect to your target.

For more details on launching Data Monitor for Linux, see [14. Using a Linux Target](#).

Starting Manually

All the same processes that are run automatically for you when you start Data Monitor from the Workbench toolbar can also be done manually from a host shell window. In addition, when starting Data Monitor manually, you have the capability to connect to multiple targets. This facilitates viewing RKMs from multiple different running targets and displaying the results on a single Data Monitor screen. For more information on RKMs, see [12. Displaying Remote Kernel Metrics](#).

VxWorks Target



CAUTION: On a VxWorks target, before entering any other commands in the Host Shell, you must type:

```
wrenv -p workbench-version
```

For example, if you are running Workbench 3.0, you would type:

```
wrenv -p workbench-3.0
```

The **wrenv** utility provides a unified way to create a command shell with a well-defined environment. It properly sets up the environment variables to allow you to start Data Monitor using the **scope** command described below.

For Data Monitor to work with your target, you must load the VxWorks target binary files to your target. Either before or after starting Data Monitor, be sure to load the following binary files to your target:

- **scopeutils.so**
- **libscope711tcp.so** or **libscope711wtx.so**
- **samlertask.so** (only for performing asynchronous sampling; see [15.4 Code Instrumentation Alternative](#), p.241)
- **vxdemo.so** (only for running the demonstration program; see [2.5 Testing Your Installation](#), p.30)

These files are located on your host at:

```
WIND_SCOPETOOLS_BASE/target/arch/targetArch
```

where **WIND_SCOPETOOLS_BASE** (an environment variable of the same name) is the root of the tree where you installed the Run-Time Analysis Tools.

All Targets

The **scope** executable is actually a wrapper script which sets up some environment variables, then calls the actual Data Monitor executable **scope-bin**, residing in the directory **scope.app**. The scope wrapper script has command line options that can be used to debug or alter the startup of Data Monitor. This directory contains plug-ins, resources, and executables used by Data Monitor, as well as the actual Data Monitor binary.

With the binary now located, the script sets the **LD_LIBRARY_PATH** path to point to the libraries needed by Data Monitor to run, and transfers control to the Data Monitor binary, located at:

```
$HOME/stethoscope/version/stethoscopy
```

where *\$HOME* is your home directory, and *version* is the current Data Monitor version.

It reads this file to determine any previously set preferences. If this file is not found, it is created.



NOTE: The contents of the `stethoscoperc` file are determined by the parameters you have set using the Data Monitor Preferences option (see the Preferences dialog description in the File menu item in [3.3 File Menu Item](#), p.43). You are strongly discouraged from trying to modify this initialization file directly.

The **scope** command has the following syntax:

```
scope[-target target] [-index n] [-verbosity level]
      [-multi targetsvr1@host1:index1:wtx targetsvr2@host2:index2
      [-errorlog filename] [-save save.ssc] [-load save.ssc]
      [-Version] [-help]
      [-tgtsvr targetServer] (VxWorks only)
      [-wtxMode] (VxWorks only)
      [-appName] (UNIX host only)
      [-library-path] (UNIX host only)
```

where the options have the following meanings:

-target *target*

Connects to the Data MonitorAPI running on the machine named *target*, where *target* can be an IP address or a target name that can be resolved to an IP address. The **-target** string is optional. If no target is specified at all, the user can choose to connect to a target at a later time.

If the target is a VxWorks target, make sure it is listed in the **HOSTS** file of your host machine. Typically, in Windows NT/2000/XP, the file is:

```
c:\Windows\system32\drivers\etc\HOSTS
```

-index *n*

Connects to the target using a specific Data Monitor channel. The index may be an integer ranging from 0 to 127. If this option is not specified, Data Monitor uses 0. A target name of form *target:n* is equivalent to **-ta target -i n**.

For example, the following commands are equivalent:

```
C:\scope -ta joshua -i 1
C:\scope -ta joshua:1
```

-verbosity *level*

Specifies the amount of diagnostic messages printed to the standard-output device. A value of 0 causes only errors to be reported. Increasing the value (in

the range of 0 - 3) increases the volume of messages. If this option is not specified, Data Monitor uses 0.

-multi *targetsvr1@host1:index1:wtx targetsvr2@host2:index2*

Enables multiple target connections. Each target specification is separated by a space, and each is formatted as:

targetsvr@host:index

where:

targetsvr is the unique name of the target server.

host is the name of the host the to which the target server is connected.

index is the Data Monitor channel number assigned to that target, in the range of 0 to 127.

If you append **:wtx** to the end of any target specification, that connection is made using WTX mode. In the absence of a **:wtx** appendix, a TCP/IP connection is made. Any other appendix will interfere with making the connection. Multiple target connections made with this argument ignore the **-index** and **-wtxMode** command line arguments described elsewhere in this list.

-errorlog *filename*

Writes verbosity messages into the file, *filename*, in addition to outputting to the log window.

-save *save.ssc*

Automatically saves the workspace state in the file **save.ssc**.

-load *save.ssc*

Reads the state saved in the file **save.ssc**.

-Version

Prints out the current Data Monitor version.

-help

Prints out the information described above.

-tgtsvr *targetServer* (VxWorks only)

When connected to a VxWorks target, specifies the WTX target-server name that manages the target. This enables the **Signal Installation** window of Data Monitor to access the target via WTX as well as the Workbench debugger to install signals automatically.

-wtxMode (VxWorks only)

Specifies that the data should be collected by the Data Monitor GUI using WTX protocol rather than TCP/IP. This option is available only for a VxWorks target, and it may not be abbreviated.

-appname (UNIX host only)

In a UNIX host, this parameter allows you to change the executable name that the script tries to run.

-library-path (UNIX host only)

In a UNIX host, this parameter allows you to change the location from which the wrapper script tries to load the **Qt** library.



NOTE: For a VxWorks target, TCP/IP communication works only if you use ScopeAPI in an RTP.

Most parameters may be abbreviated to a single letter or to as many letters as it takes to make it unique. The exceptions are noted in the descriptions above. For example, the following commands are equivalent:

```
C:\scope-> -target joshua -errorlog err.txt -verbosity 2
C:\scope-> -ta joshua -e err.txt -v 2
```



CAUTION: Your **PATH** environment variable must be set up correctly to run tools before attempting to run Data Monitor.

After starting, the Data Monitor GUI should appear in a standalone window.

Usage Notes

There are four basic steps to using the Data Monitor GUI on the host to monitor and collect data from your target application.

1. Use the Data Monitor API interface on the target to specify which data you want to be able to monitor and collect. These are known as *installed signals*, that is, signals that are registered and activated (see [Signals Definitions](#), p.19). Only *installed signals* can be collected and monitored by Data Monitor on the host. Signals can be installed manually, or automatically using the mechanism described in [Installing Signals](#), p.84.
2. Bring up the Data Monitor GUI on the host and connect to the target using the **File > Connect to Target** menu command (see [3.3.1 Connect to Target](#), p.45).



NOTE: You can connect to more than one target at a time.

3. Use the **Data Monitor Signal Manager** (see [4. Using the Signal Manager](#)) to specify which installed signals you want to collect from the target.

4. Use the **Data Monitor Plot**, **Plot XY**, **Dump Plot**, and **Monitor** windows (see Chapters 7, 8, 9, and 10, respectively) to display signals in graphs and tables. With the **Monitor** window, you can write modified signal values back out to the target. The **Plot** and **Plot XY** windows also allow you to capture and display snapshots.

Signals Definitions

Registered Signals

Initially you must let Data Monitor know a signal exists by **registering** it using the **ScopeRegisterSignal()** API call. Data Monitor cannot collect data from this signal until you Activate it. Registered signals appear in the **Signal Manager** window in the GUI, where they can be selected for activation.

Activated (or Active) signals

These are registered signals that are set up on the host by the Signal Manager (see 3.3.7 *Signal Manager*, p.49) using the API call **ScopeActivateSignal()**. Active signals then appear in the **Signals Bar** of each data-display window (see *Signals Bar*, p.26). Once activated, they are considered to be **Installed** signals and are automatically collected from the target, but they are not yet displayed in the host GUI until **Selected** in one or more of the four data-display windows, **Plot**, **Plot XY**, **Dump Plot**, or **Monitor**.

Installed signals

These are signals that are **registered** and **activated**. (They can be set up using the Data Monitor API shortcut **ScopeInstallSignal()**; see 15*Installing Signals*, p.231). A signal must be installed before the Data Monitor GUI can see it.

Selected signals

Installed signals are not displayed automatically in the GUI. You must select, from the GUI, the installed signals you want to display in the data-display windows—**Plot**, **Plot XY**, **Dump Plot**, and **Monitor**. You can select a different set of signals in each window (see *Signals Bar*, p.26).

2.4 The Data Monitor GUI

Data Monitor has four unique types of data-display windows you can use to display signal values in graphical and tabular form. By default, Data Monitor starts up with a **Plot** window displayed, but you can open any of the other window types

from the **File** menu or from the **Plots** toolbar (see [3.3 File Menu Item](#), p.43 and [3.2 Toolbars](#), p.40).

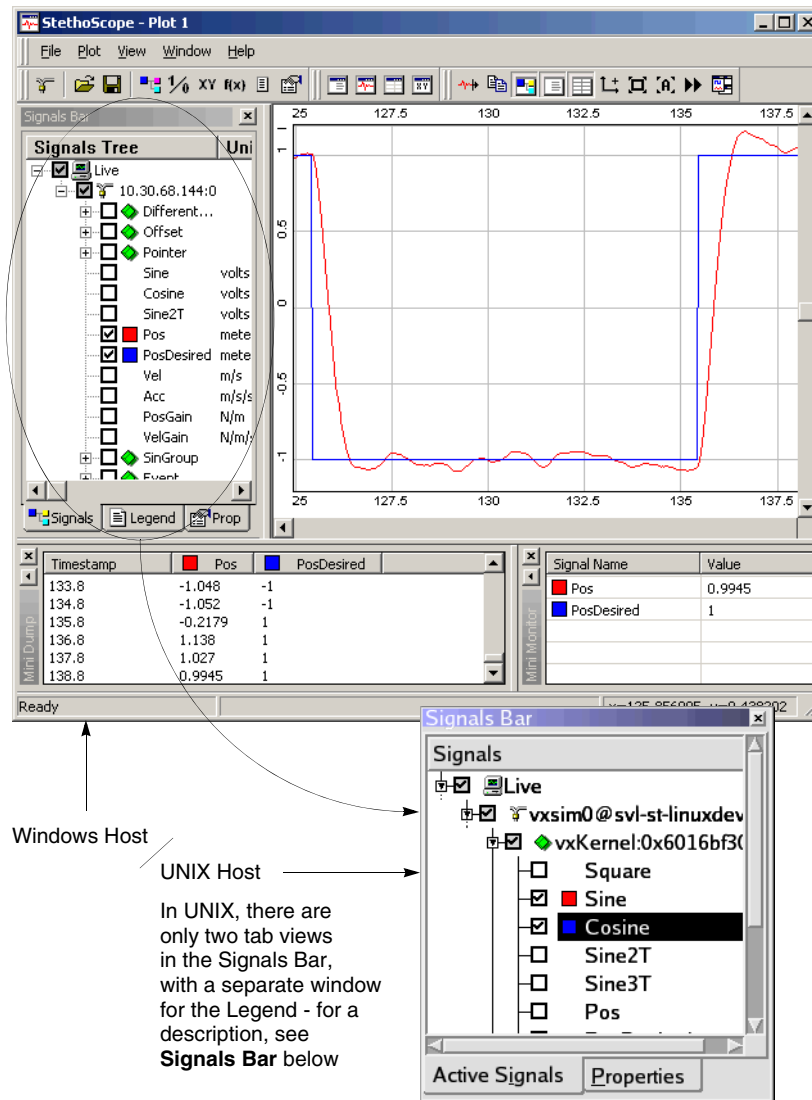
You can have multiple data-display windows of each type open at the same time. Each data-display window is independent of the others, so you can display different sets of signals in each window.

The mini-windows, panels, toolbars, and menu bar in these windows are all *dockable*, meaning you can drag them to different locations, on or off the window. You can also cause Data Monitor to save and restore these positions on future sessions (see [3.2 Toolbars](#), p.40).

Plot Window

The **Plot** window is the heart of the Data Monitor application. You can use this window to select which signals to plot, then see a color-coded plot of your selected signals over time. You can also take snapshots of plots and display them along with real-time plots for easy visual comparisons. The **Plot** window also includes mini-versions of the **Dump Plot** and **Monitor** windows (described in [Mini-Dump Window](#), p.28 and [Mini-Monitor Window](#), p.28 respectively).

A **Plot** window is displayed when you first launch Data Monitor.



The **Plot** window is the heart of the Data Monitor application. You can use this window to select which signals to plot, then see a color-coded plot of your selected signals over time. You can also take snapshots of plots and display them along with real-time plots for easy visual comparisons. The **Plot** window also includes mini-

versions of the **Dump Plot** and **Monitor** windows (described in [Mini-Dump Window](#), p.28 and [Mini-Monitor Window](#), p.28 respectively). If a Plot window is not currently open, use the **File > Plots > Plot** menu command (or the **Plot** toolbar button - see [Plots Toolbar](#), p.41) to open it.

Signals Bar

The Signals Bar in the Plot (and Plot XY) window contains tab views where you can select signals to plot, as well as set appearance and limitation properties (for the current window only). In a Windows host there is also a Legend tab view (a separate window in a UNIX host) where you can select signals to be displayed, as well as the color for plot lines. For details, see [Signals Bar](#), p.26.

Zooming

You can magnify a region to see details by zooming. The offset and scale of the plot are adjusted so that the zoomed region fills the **Plot** window.

Zooming in to a Desired Region

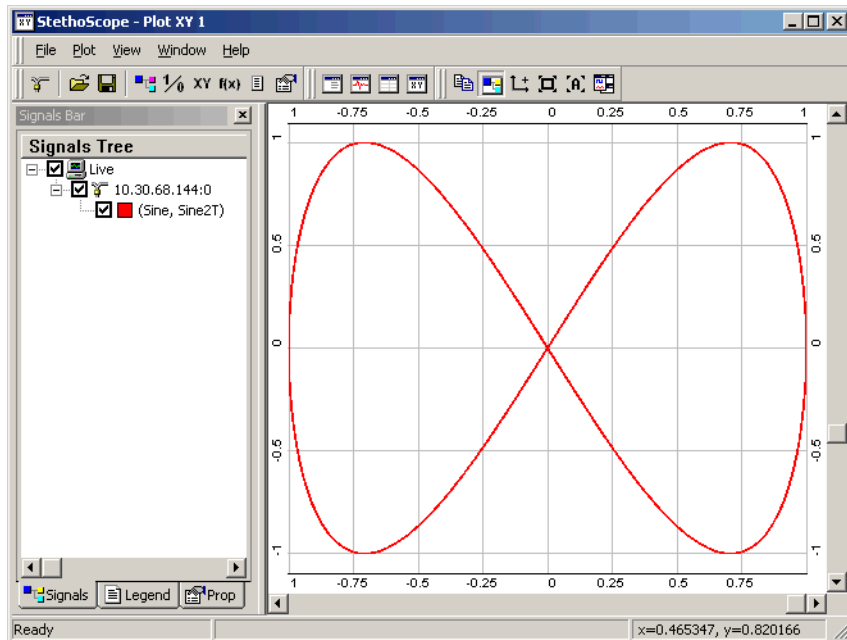
1. Press and hold the **Shift** key.
2. Click and drag the left mouse button to select a region of the plot.

The on-grid pop-up menu includes a **Previous Zoom** command, which can be used to return to the previous zoom. Note, however, that using the **Zoom to Fit** menu command (or the **Zoom to Fit** toolbar button - see [Plot Window Toolbar](#), p.41) erases the history of all zoom actions.

For a detailed description of the Plot window, see [7. The Plot Window](#).

Plot XY Window

While the **Plot** window graphs each selected signal (on the Y axis) over time (the X axis), the **Plot XY** window plots *pairs* of selected signals against each other—one signal on the X axis and the other on the Y axis.



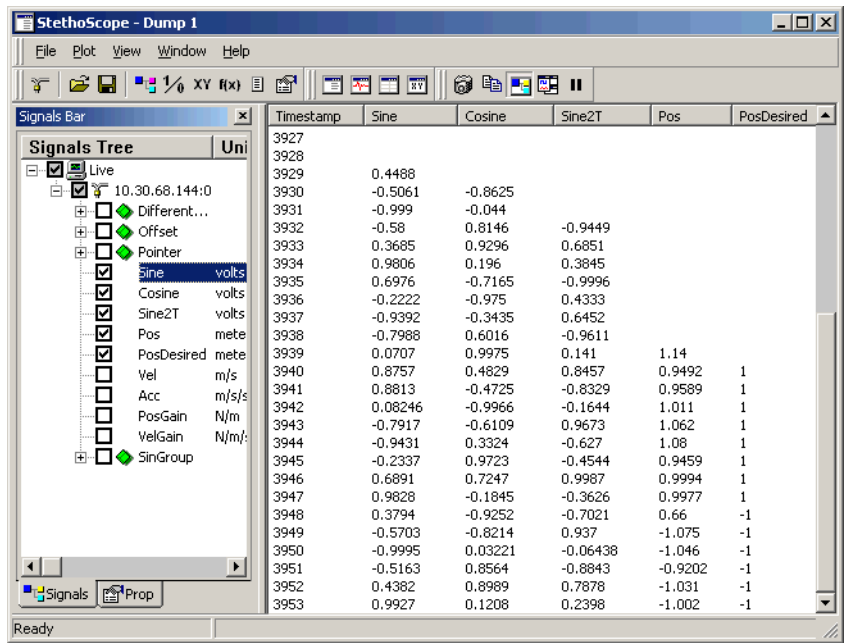
Open the **Plot XY** window with the **File > Plots > Plot XY** menu command (or the **Plot XY** toolbar button - see [Plots Toolbar](#), p.41). More than one pair of signals may be displayed at the same time. You can also take snapshots of XY plots and display them along with real-time plots for easy visual comparisons.

Note that the Plot XY Signals Bar has the same configuration as the Plot window Signals Bar (see [Signals Bar](#), p.26).

For a detailed description of the Plot XY window, see [8. The Plot XY Window](#).

Dump Plot Window

The **Dump Plot** window is used to monitor the real-time values of selected signals as they are collected from the target.



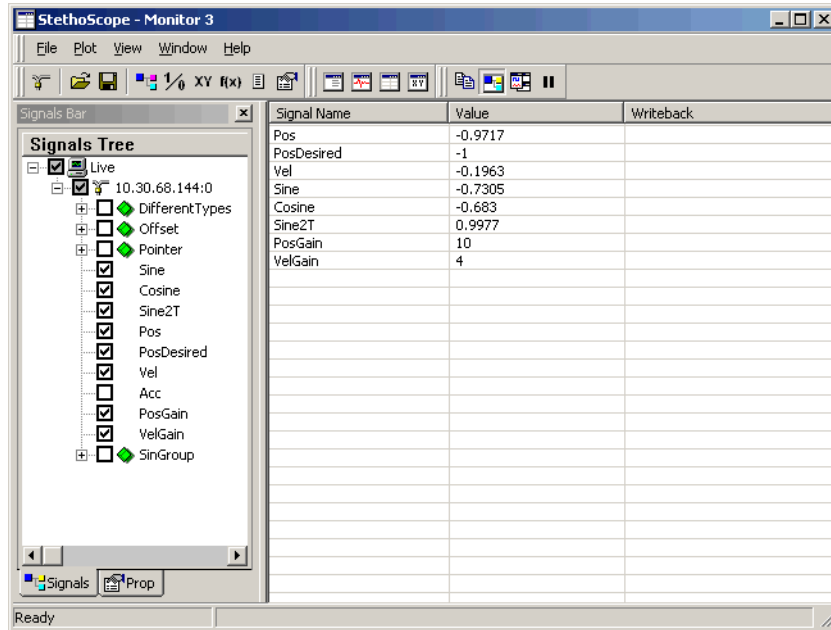
Open the Dump Plot window with the **File > Plots > Dump Plot** menu command (or the **Dump Plot** toolbar button - see [Plots Toolbar](#), p.41). It displays a simple read-only table. The first column in the table is a **Timestamp**, followed by a column for each selected signal.

Note that the Dump Plot window Signals Bar also has the same configuration as the Plot window Signals Bar (see [Signals Bar](#), p.26).

For a detailed description of the Dump Plot window, see [9. The Dump Plot Window](#).

Monitor Window

Open the **Monitor** window using the **File > Plots > Monitor** menu command (or the **Monitor** toolbar button - see [Plots Toolbar](#), p.41).



While the **Dump Plot** window displays a running history of each selected signal, the Monitor window only shows you the last sampled value of each selected signal. You can also use this window to modify the values of signals on the target.

Note that the Monitor window Signals Bar also has the same configuration as the Plot window Signals Bar (see [Signals Bar](#), p.26).

For a detailed description of the Monitor window, see [10. The Monitor Window](#).

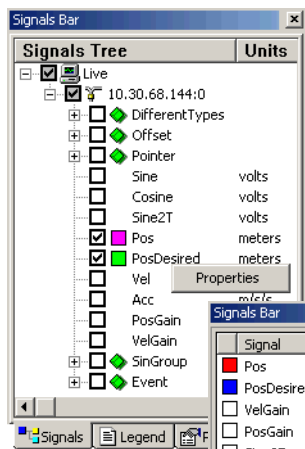
Auxiliary Data-Display Windows

Within each data-display window there are other sub-windows that display information for signal selection, and for augmenting the characteristics of your view of what is happening inside your target program. The descriptions of these additional sub-windows follows.

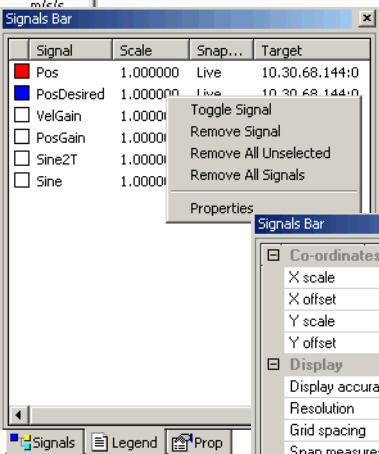
Signals Bar

By default, each of the four data-display window types contains a **Signals Bar**. If a Signals Bar is not currently displayed, you can open one using the **View > Signals Bar** menu command (or the **Signals Bar** toolbar button - see [Plot Window Toolbar](#), p.41).

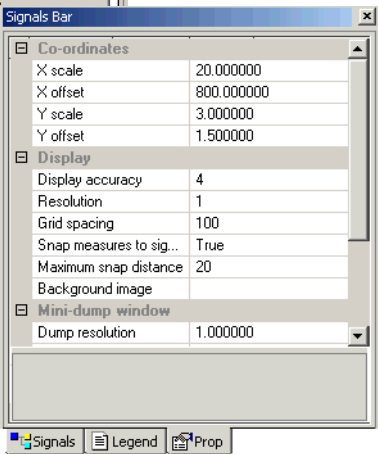
Signals Tab View



Legend Tab View



Properties Tab View

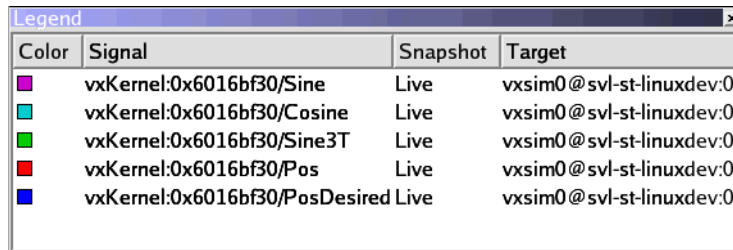


The Signals Bar has three tab views, **Signals**, **Legend** (Windows hosts only), and **Properties**.

- The **Signals** tab view is used to select which signals to monitor in the current window. It displays a signals tree, which gives you a tree-like view of the active signals for each connected target and any snapshots you have loaded. Use the check box preceding each signal to select which signals you want to display in the current window. Signals trees are described in [4. Using the Signal Manager](#). For more information, see [Signals Tab View](#), p.116.
- In Windows hosts, the **Legend** tab view shows the colors assigned to each signal. It also displays the source of each signal (**live** or **snapshot**), as well as the target IP address and Scope index. In UNIX hosts, although the data is the same, it is found in a separate **Legend** window instead of a tab view. For more information, see [Legend Window \(UNIX Hosts Only\)](#), p.122.
- The **Properties** tab view allows you to control how the signals are monitored and displayed in the window. Each type of data-display window has different properties, which are described in their respective chapters (Chapters 7, 8, 9, and 10). For more information, see [Properties Tab View](#), p.118.

Legend Window (UNIX Hosts Only)

In a UNIX host, the **Legend** window appears by default as a sub-window in the **Plot** ([7. The Plot Window](#)) and **Plot XY** ([8. The Plot XY Window](#)) windows.



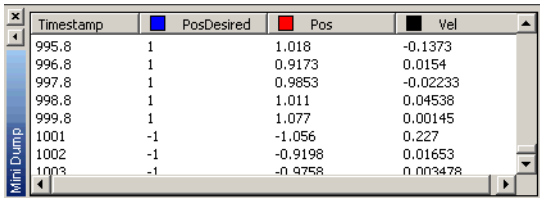
Color	Signal	Snapshot	Target
■	vxKernel:0x6016bf30/Sine	Live	vxsim0@svl-st-linuxdev:0
■	vxKernel:0x6016bf30/Cosine	Live	vxsim0@svl-st-linuxdev:0
■	vxKernel:0x6016bf30/Sine3T	Live	vxsim0@svl-st-linuxdev:0
■	vxKernel:0x6016bf30/Pos	Live	vxsim0@svl-st-linuxdev:0
■	vxKernel:0x6016bf30/PosDesired	Live	vxsim0@svl-st-linuxdev:0

This window displays the color assigned to each signal on the plot, and can also be used to select which signals to plot. If you close the window, you can open it again using the **View > Legend** menu item (or the **Legend** toolbar button - see [Plot Window Toolbar](#), p.41).

The Legend window is described in detail in [Legend Window \(UNIX Hosts Only\)](#), p.122.

Mini-Dump Window

Open the **Mini-Dump** window with the **View > MiniDump** menu command (or the **Mini-Dump** toolbar button - see [Plot Window Toolbar](#), p.41).



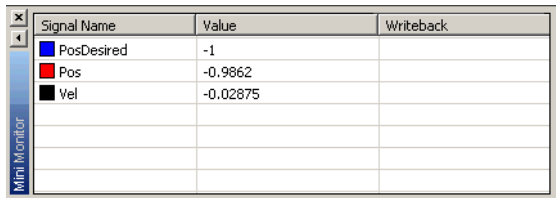
The Mini-Dump window displays a table with four columns: Timestamp, PosDesired, Pos, and Vel. The data is as follows:

Timestamp	PosDesired	Pos	Vel
995.8	1	1.018	-0.1373
996.8	1	0.9173	0.0154
997.8	1	0.9853	-0.02233
998.8	1	1.011	0.04538
999.8	1	1.077	0.00145
1001	-1	-1.056	0.227
1002	-1	-0.9198	0.01653
1003	-1	-0.9758	0.003478

Within the **Plot** window only (see [7. The Plot Window](#)), the **Mini-Dump** window shows a scaled down version of the **Dump Plot** window described in [9. The Dump Plot Window](#). Like the **Dump Plot** window, it lists the value of each signal at each sampling, letting you see a running history of selected signal values scrolling through the window with time. This mini- window initially appears at the bottom of the **Plot** window, allowing you to see numeric signal values along side the plotted signals graph. You can drag the window to any other location you wish, and it remains there until you change its location again.

Mini-Monitor Window

Like the **Mini-Dump** window described above, the **Mini-Monitor** window is opened using the **View > MiniMonitor** menu command (or the **MiniMonitor** toolbar button - see [Plot Window Toolbar](#), p.41).



The Mini-Monitor window displays a table with three columns: Signal Name, Value, and Writeback. The data is as follows:

Signal Name	Value	Writeback
PosDesired	-1	
Pos	-0.9862	
Vel	-0.02875	

This window is a scaled down version of the **Monitor** window described in [10. The Monitor Window](#). It lets you see the current value of, and modify, target data in a static but dynamically updated list format.

The modify feature (called **writeback**) is available, and is described in detail in [10.3 Writing Data to the Target](#), p.181. The writeback capability is enabled from the

Properties tab view of the Signals Bar in the Plot window, as described in [Signals Bar](#), p.116.



NOTE: **Writeback** in the **Mini-Monitor** window behaves the same as in the **Monitor** window, but enabling it here has no effect on enabling or disabling writeback in the Monitor window (see [10.3 Writing Data to the Target](#), p.181).



CAUTION: Before using the **writeback** feature, be sure to read the Warning in [10.3 Writing Data to the Target](#), p.181.

Common Window Elements

Each data-display window has certain items that are common across all the data-display window types. Some of these items are always displayed, while others can have their display toggled on or off. Some items can even be further customized.

The common window elements are:

- **Title Bar**

The title bar, on all data-display windows, indicates the name of the tool (Data Monitor), its version, and the data-display window name.

- **Menu Bar**

This bar, on all windows, contains several menu items, each with options for pertinent data manipulation functions. Menu bars are described in detail in [3.3 File Menu Item](#), p.43.

- **Toolbar**

The following toolbars are found in each window. Each is described in detail in [3.2 Toolbars](#), p.40.

- **Main** — has buttons for many of the **File** menu commands. It has the same buttons on all four data-display windows.
- **Plots** — has a button to open each of the data-display windows. It has the same buttons on all four data-display windows.
- **Plot Windows** — has different buttons for each unique data-display window. Since the **Plot Windows** toolbar buttons are different for each data-display window type, they are described in greater detail separately in each data-display window chapter (Chapters 7, 8, 9, and 10).

The three toolbars described here initially appear lined up, left to right, just below the menu bar, and separated by the docking handles at the left end of each toolbar. You can move each toolbar around to any location in the data-display window, including vertical placement, by clicking the docking handle and dragging to a new location. It remains in that location until you change it again.

- **Status Bar**

This bar, on all windows, contains several menu items, each with options for pertinent data manipulation functions. It is therefore described in detail separately in each data-display window chapter (Chapters 7, 8, 9, and 10).

2.5 Testing Your Installation

A sample demonstration target program that exercises many of the Data Monitor features is included in the Data Monitor distribution. To test your installation, and quickly and easily become familiar with Data Monitor, we strongly encourage you to run the demo program. Additional basic concepts on which Data Monitor is based are emphasized in the process of guiding you through the demo program steps. The demo program also utilizes the Data Monitor API to log program behavior.

On a VxWorks Target

For either a VxWorks or Linux platform, you can test your installation using the demonstration program **ScopeDemo**, located in:

`WIND_SCOPETOOLS_BASE/target/src/vxworks/scopedemo`

or,

`WIND_SCOPETOOLS_BASE/target/src/linux/scopedemo_linux`

where `WIND_SCOPETOOLS_BASE` (an environment variable of the same name) is the root of the tree where you installed the Run-Time Analysis Tools.

In Workbench, perform the following steps to build and execute the demonstration program:

1. Start Data Monitor running, and in the **Data Monitor Setup Options** dialog box, select **Load libraries only**, which loads the required Data Monitor libraries to your target automatically.
2. You can start the demonstration program from either of two places:
 - the Workbench GUI
 - a kernel shell

From Workbench

To start the demonstration program running from the Workbench GUI, follow these instructions:

1. In the **Remote System** view, create a target connection with an appropriate name, if one does not already exist, then connect it to the target server.
2. Right-click the connection name and select **Connect Data Monitor**, then click **OK** in the **Connect to Target** dialog box to accept the default connection parameters.

Note that the Data Monitor GUI opens in a standalone window mode.

Verify that the status message in the **Analysis Console** view is:

Connected to *target*

If this message does not appear, check the Analysis Console view for error messages.

3. Build the Data Monitor example program vxdemo.c following these instructions:
 - a. Right-click anywhere in the **Project Explorer** view and select **New**, then **Example** to open the **New Example** dialog box.
 - b. Select **VxWorks Downloadable Kernel Module Sample Project** in the **New Example** dialog box that opens, then click **Next**.
 - c. Select **The Data Monitor Demonstration Program** in the **New Project Sample** dialog box that opens, then click **Finish** to complete the project creation.

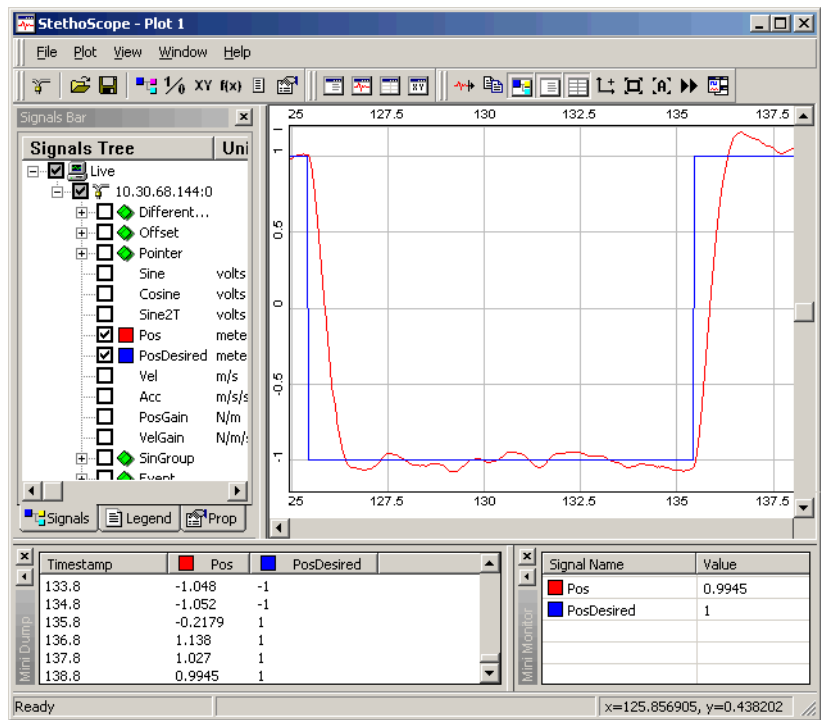
Notice that a new **scopedemo** node now appears in the Project Explorer view. Next you need to build the scopedemo.c program.

- d. In the Project Explorer view, expand the top (scopedemo) node, then right-click the scopedemo (scopedemo.out) node and select **Rebuild Project** to build the binary files.

This program builds rather quickly, but you can follow the build progress in the **Build Console** view, as well as the progress meter in the **Build Projects** dialog box.

4. When the program has successfully built, execute it by following these steps.
 - a. In the Project Explorer view, right-click the **scopedemo.out** node and select **Download**, then click **OK** in the **Download** dialog box that opens to download the executable files.
 - b. In the Project Explorer view again, right-click the **scopedemo.out** node and select **Run Kernel Task**.
 - c. In the **Run** dialog box that opens, in the **Kernel Task to Run** group, click **Browse** in the **Entry Point** field, and select **Downloads > scopedemo.out > ScopeDemo** as the binary files to be loaded, then click **Apply**.
 - d. In the **Arguments** field, enter the appropriate desired parameters, if any (see Step 2. in [From a Kernel Shell](#) below).
 - e. Click **Run** to start the **ScopeDemo** example program executing.

A Data Monitor **Plot** window similar to this one should appear on your screen.



In this example, the **Pos** and **PosDesired** signals have been selected for display. The status bar at the bottom of the window shows the connection status. Under normal conditions, it should display **Ready**.



NOTE: Again, the Signals Tree contains three tab views in Windows hosts, but only two tabs in UNIX hosts. For details, see [Legend Tab View \(Windows Hosts Only\)](#), p.117.

From a Kernel Shell

To start the demo program running from a kernel shell, use the following instructions:

1. In the kernel shell (the Data Monitor simulator in this case), navigate to the demo directory and load the demo file vxdemo.so with these instructions:

```
cd /WIND_SCOPETOOLS_BASE/target/arch/simntVx6.6gcc4.1.2
```

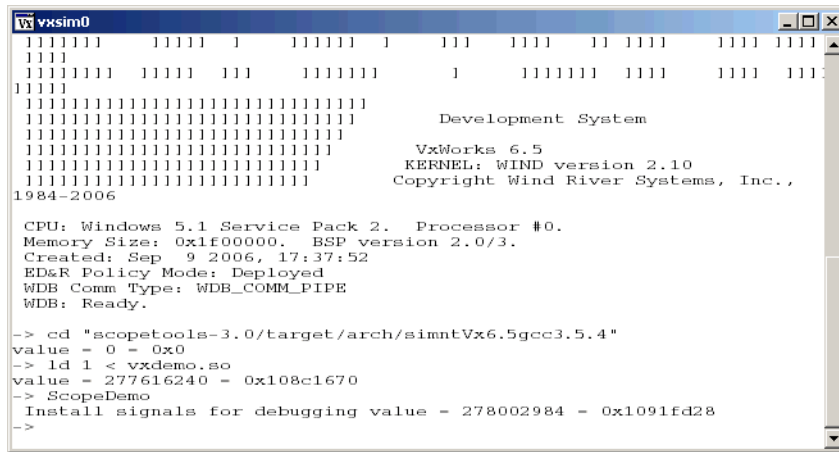
```
ld 1 < vxdemo.so
```

where `WIND_SCOPETOOLS_BASE` (an environment variable of the same name) is the root of the tree where you installed the Run-Time Analysis Tools.

2. Start the demo program by typing the following (arguments are optional):

```
ScopeDemo [[useAuxClk], scopeIndex], verbosity]
```

The kernel shell should display results as shown here.



```
vxsim0
|||||  ||||| 1  ||||| 1  |||  ||||  || |||||  ||||  |||||
|||||
|||||||  ||||| |||  ||||| 1  |||||  ||||  ||||  |||
|||||
|||||||||||||||||||||||||||||||||||||
||||||||||||||||||||||||||||||||||||| Development System
|||||||||||||||||||||||||||||||||||||
||||||||||||||||||||||||||||||||||||| VxWorks 6.5
||||||||||||||||||||||||||||||||||||| KERNEL: WIND version 2.10
||||||||||||||||||||||||||||||||||||| Copyright Wind River Systems, Inc.,
1984-2006

CPU: Windows 5.1 Service Pack 2. Processor #0.
Memory Size: 0x1f00000. BSP version 2.0/3.
Created: Sep 9 2006, 17:37:52
ED&R Policy Mode: Deployed
WDB Comm Type: WDB_COMM_PIPE
WDB: Ready.

-> cd "scopetools-3.0/target/arch/simntVx6.5gcc3.5.4"
value = 0 = 0x0
-> ld 1 < vxdemo.so
value = 277616240 = 0x108c1670
-> ScopeDemo
Install signals for debugging value = 278002984 = 0x1091fd28
->
```

➔ **NOTE:** If you run **ScopeDemo** without any parameters, then **ScopeDemo** uses **scopeIndex=0**. In this case, be sure to specify **0** for **scopeIndex** in the **Setup Options** dialog box (see [Starting Automatically](#), p.11) when you start the Data Monitor GUI. Or if you start the Data Monitor GUI first with a non-zero **scopeIndex**, be sure to start the **ScopeDemo** with that same **scopeIndex** value (note the argument position in Step 2. above).

➔ **NOTE:** On VxWorks, if you do not have a target server running, you must create it first. For details on how to configure and start a target server, refer to *Wind River Workbench User's Guide*.

On a Linux Target

A copy of the same demo program is also available to be compiled and run on a Linux target. Copy the Linux program to your Linux target, then follow the Linux counterpart directions outlined for VxWorks above. Output from the Linux demo program appears in the Data Monitor GUI essentially the same as it does for the VxWorks version.

Viewing the Signals

The demonstration program generates several sample signals that can be viewed as follows:

1. If the **Plot** window does not include a **Signals Bar**, open one using the **View > Signals Bar** command (or the **Signals Bar** toolbar button - see [Plot Window Toolbar](#), p.41).
2. Select the signals you want to plot by using the **Signals Tree** (see [4. Using the Signal Manager](#)).



CAUTION: It is a known problem that if you select **Events** to be plotted simultaneously with signals, the resulting traces displayed over a period of time will begin to diverge, and the integrity of the plot becomes compromised. Do not try to plot these two data types on the same graph.

Exploring the Demo Capabilities

Try each of the following Data Monitor features, consulting the referenced manual section if you need help. The demo allows you to:

- Display other signals by clicking on signal entries in the **Signals Tree** ([Signals Bar](#), p.26).
- Take a **Snapshot** from the **Plot** window, and save it (Sections [11.2 Utilizing Snapshots](#), p.185 and [Saving Snapshots](#), p.187).
- Export **Snapshot** data ([Exporting Snapshots in MATLAB and MATRXXX](#), p.193).
- Zoom in and out (**Shift** key + left mouse button in **Plot** screen) ([Zooming](#), p.76).
- Pan the viewing region (click and drag the left mouse button to move) ([Panning](#), p.77).

- Take measurements (**Ctrl** key + left mouse button in **Plot** screen) (*On-grid Measurements*, p.78).
- Calculate derived signals (**File > Derived Signals**) (*6. Derived Signals*).
- View and modify variables. (Select **Pos** and **PosGain** signals in a **Monitor** window with **Writeback** turned on) (Sections *10.2 Monitor Window Tour*, p.174 and *10.4 Setting New Monitor Window Preferences*, p.182).
- Try X vs. Y plotting (**Plot XY** from the **File > Plots** menu) (*8. The Plot XY Window*).
- Display numeric data (**Dump Plot** from the **File > Plots** menu) (*9. The Dump Plot Window*).
- Set some triggers (**File > Triggering**) (*5. Triggering*).

There are additional features you can try. Note that the signals produced are simple sine waves, and in addition, the demo program also produces a simple simulation of a control system.

Automatic Signal Management (VxWorks Only)

You can install additional signals automatically from the VxWorks command shell (System Viewer or kernel shell). You can use the following procedure to watch your own signals.

A simple command to load a signal is **ScopeInstallSignal()**. The calling syntax is:

```
ScopeInstallSignal
char *name,          /* the string to be displayed by Data Monitor */
char *units,         /* the units of the signal */
void *ptrToVar,      /* a pointer to your variable */
void char *type,     /* the type, e.g. "float", "int" */
int index)          /* The scope index (defaults to 0) */
```

As an example, the demo program contains a static variable declared as:

```
float Kp = 10;
```

To install this signal from the VxWorks shell, type:

```
-> ScopeInstallSignal("Kp", "n/a", &Kp, "float", 6)
```

More sophisticated installations of variables referenced by pointers, offsets, and so forth, can also be done easily.

Some other fun things to play with in the VxWorks shell when running the demo are:

```
-> ScopeRemoveMultipleSignals("sin", 6)
-> Kp = (float) 100.0
```

Executing the first command should remove all signals that start with **sin**. Look at **Pos** and **PosDesired** in the **Plot** window to see the effect of changing the value of **Kp**.

3

Data Monitor Features

3.1	Introduction	39
3.2	Toolbars	40
3.3	File Menu Item	43
3.4	Menu Bar	69
3.5	Pop-up Menus	71
3.6	Screen Operations	76
3.7	Status Bar	78

3.1 Introduction

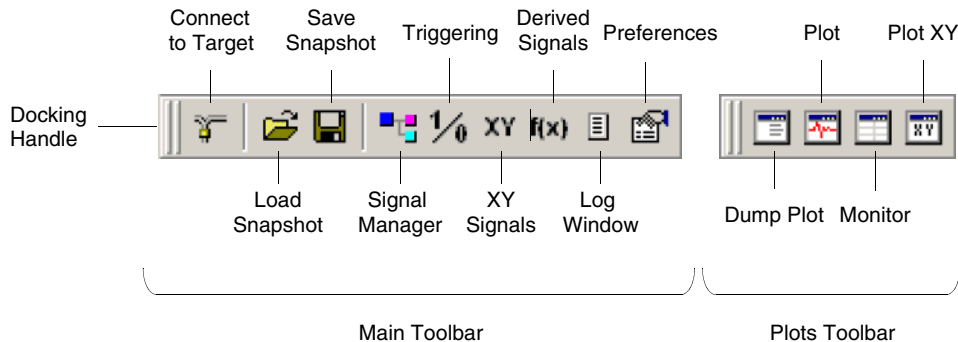
This chapter describes the major Wind River Data Monitor features. These features are available through the **File** menu item, and are common to all four plot windows (described in [2.4 The Data Monitor GUI](#), p.19) of the Data Monitor graphical user interface (GUI). Each of the features is described in detail in the sections that follow.

3.2 Toolbars

The toolbars appearing at the top of each data-display window actually consists of three separate toolbars, each offering quick access to commonly used menu commands. Place your mouse pointer over each toolbar button to see a **ToolTip** that shows you the action of each button.

Each toolbar is dockable, which means you can move it to another location, on or off the window, simply by dragging it. (The menu bar is also dockable.) Each toolbar can be independently displayed or hidden using the **View** menu item. Toolbars that you drag off the screen can be restored again at any time from the View menu.

The following figure shows the first two toolbars: **Main** and **Plots**. These toolbars are the same in all data-display windows.



Main Toolbar

The buttons on this toolbar control the various Data Monitor functions. The corresponding menu commands are in bold.










File > Connect to Target

Selects a target and connects it to your host GUI, described in [3. Data Monitor Features](#).







File > Load Snapshot

Loads a previously saved snapshot file, described in [11. Working with Snapshots](#).

-  **File > Save Snapshot**
Saves a snapshot to a file, described in [11. Working with Snapshots](#).
-  **File > Signal Manager**
Opens the **Signal Manager** window, described in [4. Using the Signal Manager](#).
-  **File > Triggering**
Opens the **Triggering** window, described in [5. Triggering](#).
-  **File > XY Signals**
Opens the **XY Signals** dialog box, described in [8. The Plot XY Window](#).
-  **File > Derived Signals**
Opens the **Derived Signals** dialog box, described in [6. Derived Signals](#).
-  **File > Log Window**
Opens the **Trace Log** window, described in [3.3.11 Trace Log Window](#), p.53.
-  **File > Preferences**
Opens the **Preferences** dialog box, described in [3.3.12 Preferences](#), p.53.

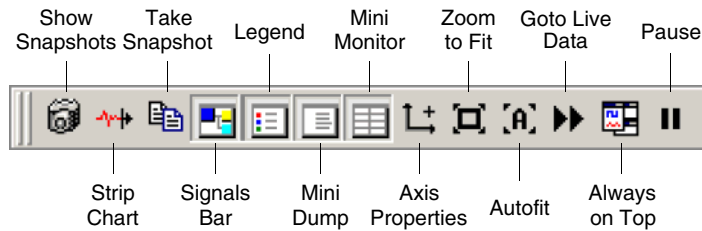
Plots Toolbar

The buttons on this toolbar are used to create new data-display window. The corresponding menu commands are in bold.

-  **File > Plots > Dump Plot**
Creates a new **Dump Plot** window, described in [9. The Dump Plot Window](#).
-  **File > Plots > Plot**
Creates a new **Plot** window, described in [7. The Plot Window](#).
-  **File > Plots > Monitor**
Creates a new **Monitor** window, described in [10. The Monitor Window](#).
-  **File > Plots > Plot XY**
Creates a new **Plot XY** window, described in [8. The Plot XY Window](#).

Plot Window Toolbar

The buttons on the third toolbar are shown below. The toolbar shown here is a super-set of buttons appearing on the toolbar in each of the four data-display windows (**Plot**, **Plot XY**, **Dump Plot**, and **Monitor**).



This section briefly describes each toolbar button. For more details on the command descriptions, refer to [3.3 File Menu Item](#), p.43.



View > Show Snapshots

Available only in the **Dump Plot** window, this command controls what is displayed in the **SignalsTree** list. When selected, only **snapshots** appear in the Dump Plot window SignalsTree list, and when unselected, only the **live data** buffer appears. For details, see [Loading Snapshots](#), p.193.



Plot > Strip Chart

Changes the display so that it presents a continuous, scrolling plot (instead of repainting the plot every 10 seconds). This button (and command) is only available in the **Plot** window. For details, see [Strip Chart](#), p.122.



Plot > Take Snapshot

Saves a copy of all the active signals. You can display the snapshot in the **Plot** and **Plot XY** windows along with real-time data. For details, see [Taking Snapshots](#), p.186.



View > Signals Bar

Creates a Signals Bar panel in the window. For details, see [2.4 The Data Monitor GUI](#), p.19.



View > Legend (UNIX hosts only)

Opens a **Legend** window in the GUI. For details, see [Legend Window \(UNIX Hosts Only\)](#), p.122.



View > Mini-Dump

Creates a Mini-Dump window within the **Plot** window. This is simply a smaller version of the separate Dump Plot window. It is created as a sub-window in a default location in the Plot window, but you can dock it elsewhere on the screen. This button (and command) is only available in the Plot window. For details, see [Mini-Dump Window](#), p.28.

**View > Mini-Monitor**

Creates a Mini-Monitor window within the **Plot** window. This is simply a smaller version of the separate Monitor window. It is created as a sub-window in a default location in the Plot window, but you can dock it elsewhere on the screen. This button (and command) is only available in the Plot window. For details, see [Mini-Monitor Window](#), p.28.

**View > Axis Properties** (Windows hosts only)

Opens the Axis Properties dialog box, where you can add new axes, and edit the properties for any existing axis. For details, see [7.4 Axis Properties Dialog Box \(Windows Hosts Only\)](#), p.128.

**View > Zoom to Fit**

Changes the scales and offsets so that all the signals fit and take up the entire plot window. This button (and command) is only available in the Plot and Plot XY windows. For details, see [View Menu Item](#), p.113.

**View > Auto Fit**

The plot automatically zooms to fit when a signal goes off the screen. This button (and command) is only available in the Plot and Plot XY windows. For details, see [View Menu Item](#), p.113.

**View > Goto Live Data**

Returns to plotting live data while viewing a snapshot in the Signals Bar. For details, see [View Menu Item](#), p.113

**Window > Always on Top**

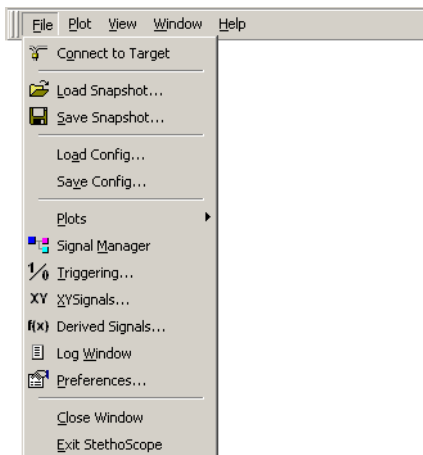
Keeps the window as the topmost display on your desktop. For details, see [Window Menu Item \(Windows Hosts Only\)](#), p.70.

**View > Pause**

Stops updates to the table only in the Dump Plot window. It does not stop data collection, but merely stops new data from appearing in the Dump Plot table. To resume normal display, unselect the button. For details, see [View Menu Item](#), p.166.

3.3 File Menu Item

Most of the Data Monitor common functionality is accessible directly through the **File** menu item on the menu bar.



The menu bar is **dockable**, which means you can move it to another location, on or off the window, simply by clicking on the docking handle and dragging the menu to the new location. Each command in this **File** menu is found in all four of the Data Monitor data-display windows in both Windows and UNIX hosts.

These commands are listed and described in detail in the sub-sections that follow.

- [Connect to Target](#), p.45
- [Load Snapshot](#), p.45
- [Save Snapshot](#), p.45
- [Load Config](#), p.46
- [Save Config](#), p.46
- [Plots](#), p.48
- [Signal Manager](#), p.49
- [Triggering](#), p.50
- [XY Signals](#), p.51
- [Derived Signals](#), p.52
- [Trace Log Window](#), p.53
- [Preferences](#), p.53
- [Close Window](#), p.69
- [Exit Data Monitor](#), p.69

3.3.1 Connect to Target

After starting the Data Monitor GUI, you need to connect to your target. Select the **Connect to Target** menu command from the **File** menu (or use the **Connect to Target** toolbar button - see [Main Toolbar](#), p.40) to open the Data Monitor Setup Options dialog box where you can create a connection to a target. This dialog box is the same in both Windows and Unix hosts.

The **Data Monitor Setup Options** dialog box opens with initial values as passed in from the command line or the GUI (see the figures at [VxWorks Data Monitor Setup Options Dialog Box](#), p.12, or [Linux Data Monitor Setup Options Dialog Box](#), p.13, as appropriate). Use this dialog box to enter any desired connection parameters, then click **OK** to connect to your VxWorks target server, or Linux target.

If you have difficulty connecting, check the **Log** window (see [Trace Log Window](#), p.53) for status or error messages.



NOTE: The Connect to Target dialog box can also be called repeatedly to establish **multiple target connections**, if desired (or you can start Data Monitor manually and use the **-multi** command; see [Starting Manually](#), p.14).

3.3.2 Load Snapshot

Snapshots are described in detail in [11. Working with Snapshots](#). Snapshots that have been saved to disk in the native Data Monitor format (**.ss7** extension) can be reloaded for viewing in any of the plot windows using the **File > Load Snapshot** menu command (or the **Load Snapshot** toolbar button - see [Main Toolbar](#), p.40).

3.3.3 Save Snapshot

Snapshots are created with the **Plot > Take Snapshot** menu command (or the **Take Snapshot** toolbar button - see [Plot Window Toolbar](#), p.41). They can be saved to disk for future reference and reloading, using the **File > Save Snapshot** menu command (or the **Save Snapshot** toolbar button - see [Main Toolbar](#), p.40). For information on this process, see [Saving Snapshots](#), p.187.

3.3.4 Load Config

You can load the configuration parameters set up in a previous Data Monitor session, and saved with the **File > Save Config** menu command (described in [3.3.5 Save Config](#), p.46), using the **File > Load Config** menu command.

To load configuration parameters, do the following:

1. Select the **File > Load Config** menu command to bring up the **Open** dialog box.
2. Navigate to the pathname containing the file you want to load. The default directory is the same directory where you installed Data Monitor.
3. Select or enter the name of the desired file in the **Filename** field of the dialog box.
4. Click **Open** to open the file and load the configuration parameters.

If signals that were active when the configuration was saved are not present when it is restored, (that is, they have not been activated or installed via **ScopeInstallSignal()**), then they will not displayed in the window. However, the window displays them as soon as they become available in the current Data Monitor session.

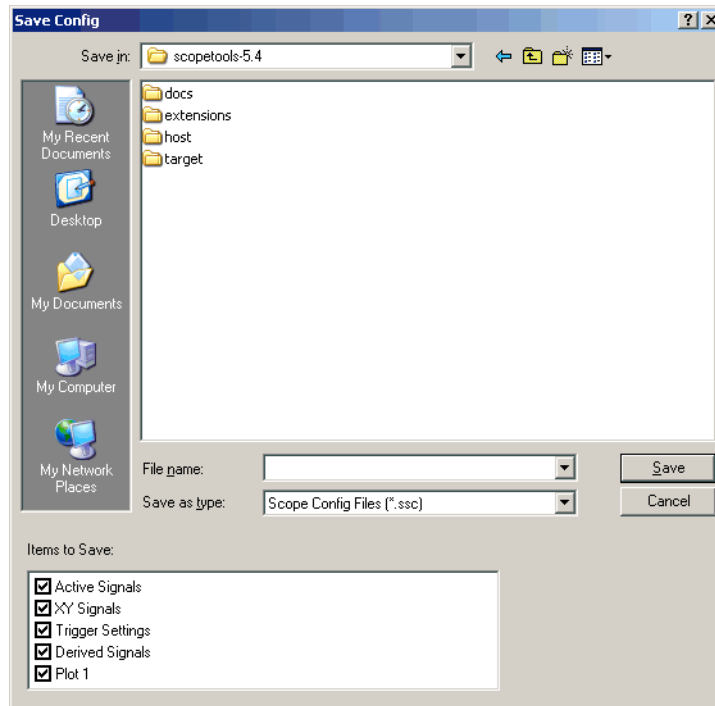
3.3.5 Save Config

There are configuration variables you can modify to customize the appearance and behavior of your Data Monitor GUI. Whenever you exit Data Monitor, the current settings of these configuration variables are saved in a default configuration file, which is then reloaded the next time you start Data Monitor. You may find a relatively constant set of configuration parameters that meet your needs, and automatic saving and reloading is satisfactory.

In some cases, however, you may find yourself changing the configuration of the Data Monitor GUI substantially for different projects or environments. In these cases you can save the current configuration parameters to a file which can then be reloaded when you work on that project.

To save current configuration parameters at any time, do the following:

1. Select the **File > Save Config** menu command to open the **Save Config** dialog box.



2. Navigate to the pathname where you want to store the file to be saved. The default directory is the same directory where you installed Data Monitor.
3. Select or enter a filename in the **Filename** field of the dialog box.
4. The check boxes at the bottom of the **Save Configuration** window allow you to save selected portions of the state of the current Data Monitor session. Select from the **Items to save** list by checking the corresponding check boxes for the following items:
 - **Active Signals** — Saves the list of currently activated signals. On reload, if any of the signals in this list have not been registered, they appear grayed-out in the Signal Manager window. For details on the Signal Manager, see [4. Using the Signal Manager](#), and for registering signals, see [Registering and Activating Signals](#), p.249.
 - **XY Signals** — Saves the list of currently defined **XY Signals**. These signals may not appear immediately on reload if any of the component signals are not yet active. They appear when all their components become available.

- **Trigger Settings** — Saves all the triggering and sampling parameters.
 - **Derived Signals** — Saves the list of currently defined derived signals. The derived signals may not appear immediately on reload if any of the component signals are not yet active. They appear when all their components become available.
 - **Plot *n*, Plot_XY *n*, Monitor *n*, Dump *n*** — Saves the state of the named data-display windows (where *n* is the unique number of that type window in the title bar). When each of these files is loaded, a new window is opened if none with the saved name is currently open. If a data-display window with that name already exists, then its state is adjusted to match the saved information. The state of a window includes the state of a window includes the list of selected signals being displayed in the window, as well as the size and shape of the window.
5. Click **Save** to save the configuration. By default, the filename has the extension **.ssc**.

This feature allows you to develop a library of saved Data Monitor configurations. For example, you might want to create the following configuration files:

- **position.ssc**—containing only the state of one **Plot** window that has been set up exactly the way you like it for displaying your position sensors.
- **derived.ssc**—containing a useful set of derived signals, such as a scaled variable or the difference of two signals.

3.3.6 Plots

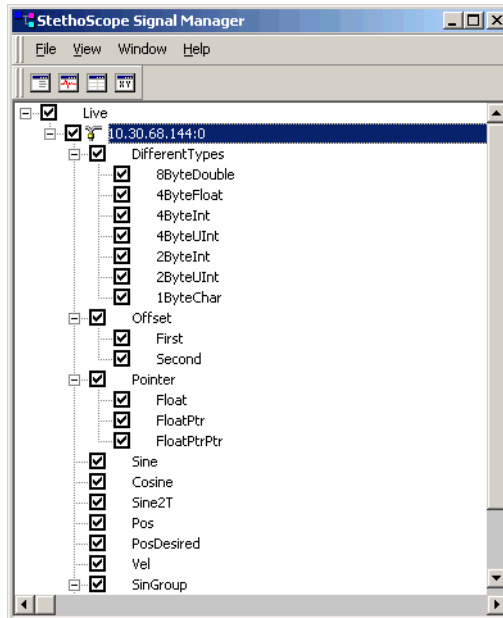
The drop-down menu opened with this command lists the four basic Data Monitor data-display windows. Selecting one opens that window (or another occurrence of that window if one is already open). You can have multiple data-display windows open at the same time, and each can display different signals or snapshots.

Detailed information for each plot window is found in [7. The Plot Window](#), [8. The Plot XY Window](#), [9. The Dump Plot Window](#), and [10. The Monitor Window](#) respectively.

Before using data-display windows, it may help to understand how and when data is collected from the target; for this see [5. Triggering](#).

3.3.7 Signal Manager

Open the **Data Monitor Signal Manager** window using the **File > Signal Manager** command (or the **Signal Manager** toolbar button - see [Main Toolbar](#), p.40).



This window allows you to select which signals are collected from each target. Only these active signals can be monitored in any of the four types of data-display windows. The Signal Manager presents a tree-like view of each target.

The Signal Manager can see all the signals that were installed (that is, registered and activated; see [Registering and Activating Signals](#), p.249) on the target. If your target has many installed signals, the **Signals Trees** in the data-display windows can become very cluttered. You can use the Signal Manager to dynamically filter out installed signals that you do not want to collect or have appear in Signals Trees.

If you do not use the Signal Manager, then by default, all installed signals on the loaded targets are collected and available for monitoring.

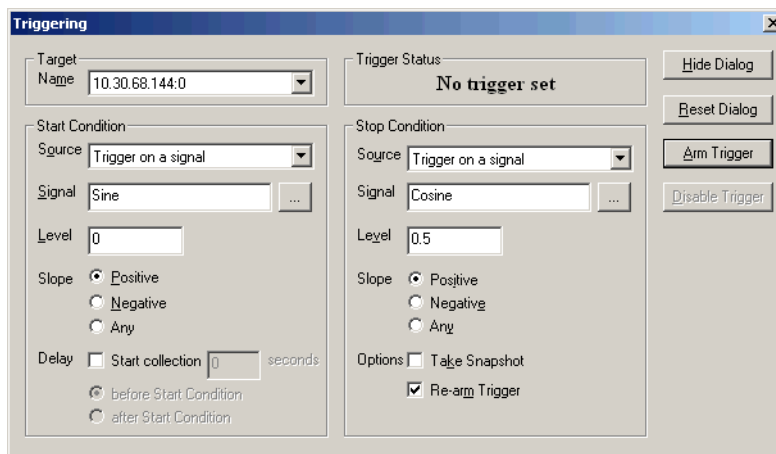
Unlike the other types of Data Monitor windows, you only need (and can only have) one Signal Manager window, so what you do in this window impacts what

you see in all the other types of Data Monitor data-display windows. Changes made in the Signal Manager are immediately propagated to all data-display windows. For example, if you are monitoring a signal in a **Plot** window and a **Plot XY** window, and you make the signal inactive in the Signal Manager window, it is removed from both your Plot and Plot XY windows immediately. Similarly, if you activate new signals in the Signal Manager window, they are added to both the Plot and Plot XY window Signals Trees (and for any other open data-display windows), where you can choose to select the new signals or not. For more information, see [4. Using the Signal Manager](#).

3.3.8 Triggering

The Data Monitor API module on the target is responsible for handling periodically collected data (or samples) and sporadically collected data, or events. The **Triggering** facility in Data Monitor provides control over when and how often the samples are collected. The triggering facility supports a single trigger at any given time.

Open the **Triggering** dialog box with the **File > Triggering** menu command (or the **Triggering** toolbar button - see [Main Toolbar](#), p.40).



Triggering is disabled when the Triggering dialog box is not open, or the dialog box is open but the trigger has not yet been armed. In this state, calls to `ScopeCollectSignals()` result in data being collected normally.

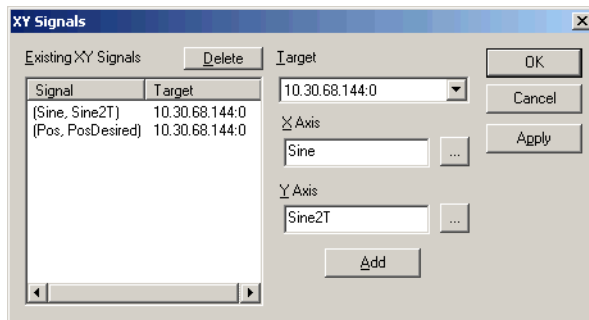
When the Triggering dialog box has been opened and the trigger is armed with valid start and stop conditions, all calls to **ScopeCollectSignals()** from that point on return without collecting any data. In this armed state (that is, the trigger is valid, but has not fired), sampled signals are not plotted in any of the GUI windows, although event data continues to be plotted as before. Collection and plotting of samples begins when the **Start Condition** is met (that is, the trigger fires). Data continues to be plotted on the GUI until the **Stop Condition** occurs. At that point the trigger is then either disabled or rearmed depending on the **Rearm** option specified in the Triggering dialog box.

To learn more about how triggers are set and used, see [5. Triggering](#).

3.3.9 XY Signals

The **Plot XY** window is used to graph pairs of signals against each other. Signal selection for Plot XY windows differs from signal selection for the other types of data-display windows in that each plotted data line is composed of two signals. You must create these XY signal pairs using the **XY Signals** dialog box before you see them in the Plot XY window **Signals Tree**. They appear in a separate branch of that tree.

Open the **XY Signals** dialog box using this command (or the **XY Signals** toolbar button - see [Main Toolbar](#), p.40).

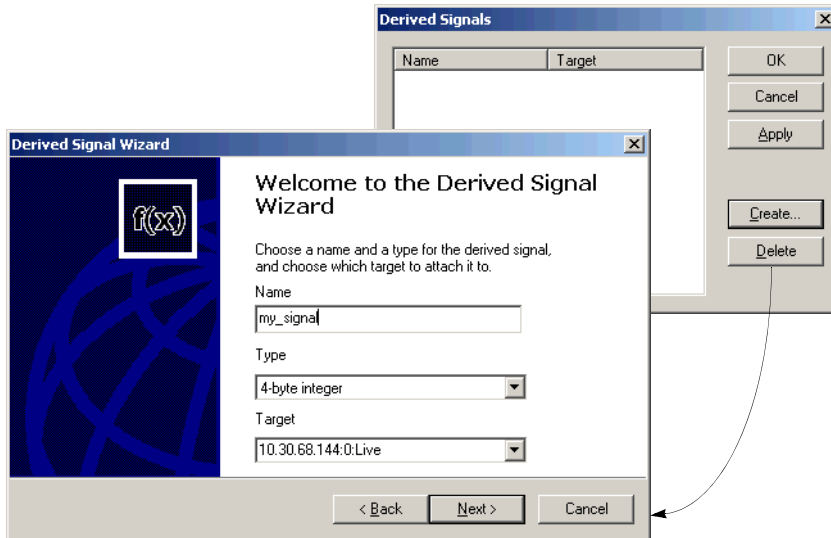


For details on how to create signal pairs with the XY Signals dialog box, and display those signal pairs using the Plot XY window, see [8. The Plot XY Window](#).

3.3.10 Derived Signals

A derived signal is a new signal you create whose value is computed by mathematical operations on other, existing signals. Derived signals are calculated by Data Monitor on the host in real-time, but not displayed directly from your real-time system. The derived-signals facility provides a simple means of scaling and offsetting signals, plotting differences and ratios of signals, and so forth.

You must create a derived signal using the **Derived Signals** dialog box before it appears in a separate branch in the **Signals Tree** of the **Plot**, **Dump Plot**, and **Monitor** windows. Open the Derived Signals dialog box using the **File > Derived Signals** menu command (or the **Derived Signals** toolbar button - see [Main Toolbar](#), p.40).



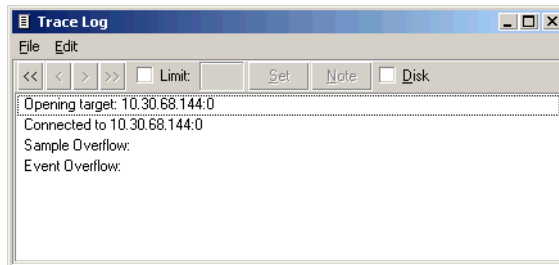
This dialog box utilizes a series of **Derived Signal Wizard** dialog boxes, also shown in the figure above, to guide you through the process.

For details of how to create derived signals, see [6. Derived Signals](#).

3.3.11 Trace Log Window

Data Monitor records events in a log file, which you can display in the **Trace Log** window, as well as save to disk. The types of events recorded in the log file depend on the **Verbosity** setting selected when Data Monitor was started. With each increase in verbosity, more events are included in the log file. **Verbosity** is a command-line option (see [Starting Automatically](#), p.11). Verbosity can also be set on a per-target basis with the **Connect to Target** window (see [3.3.1 Connect to Target](#), p.45).

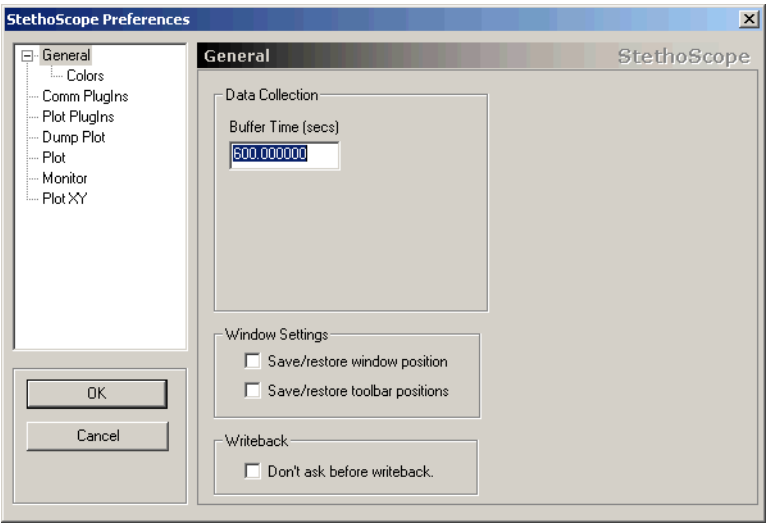
Open the **Trace Log** window using the **File > Log Window** command (or the **Log Window** toolbar button - see [Main Toolbar](#), p.40), or by double-clicking anywhere in the error message area of the status bar for any given data-display window (described in Chapters 7, 8, 9, and 10).



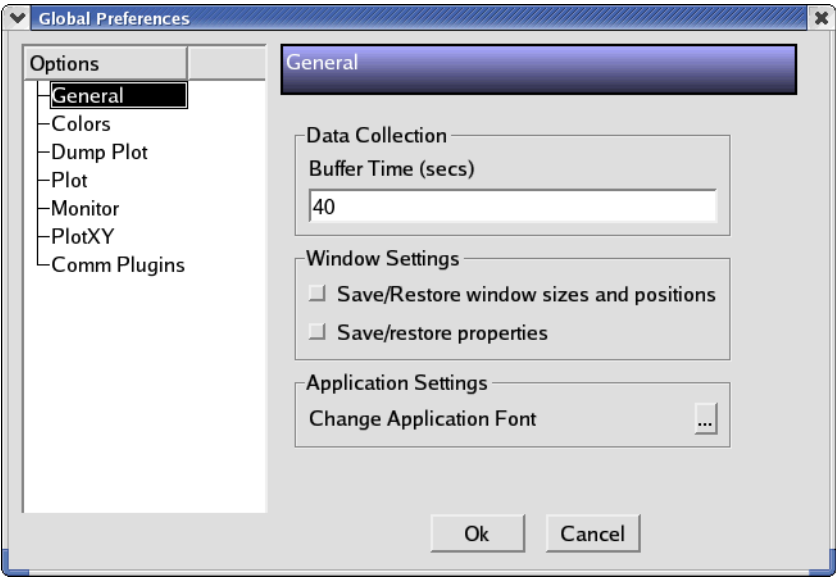
3.3.12 Preferences

Open the **Data Monitor Preferences** dialog box using the **File > Preferences** command (or the **Preferences** toolbar button - see [Main Toolbar](#), p.40). Note that this dialog box has a few significant differences between the Windows and UNIX host versions. Therefore both versions are illustrated here, and their descriptions are noted. The order of menu descriptions varies, but all items from both dialog boxes are included.

Windows Host



UNIX Host



This dialog box allows you to set preferences (defaults) for the Data Monitor GUI when it starts and when new data-display windows are created. Preferences are different than properties—**preferences** are the values used when new data-display windows are created, whereas **properties** apply only to a specific data-display window that is currently open. In other words, preferences are the default properties used for new data-display windows. Then, once a data-display window is open, you can change its properties using the **Properties** menu command as described in the chapter for each specific data-display window.

The left-most panel displays the various types of preference selections available. The following sub-sections describe the options available for each of these preference views.

1. General View

The General view, shown above, allows you to control some aspects of data collection and display, as well as whether window and toolbar positions are maintained when you exit and restart Data Monitor.

The parameters in the **Data Collection** panel are:

- **Buffer Time (secs)** — The buffer time indicates how many seconds of data to show in the plot before refreshing. This is only used when **not** in **Strip Chart** mode, and only in the **Plot** and **Plot XY** windows. This value is seen on the plot as the width of the grid (X-axis). The default is 1 second.

The parameters in the **Window Settings** panel are:

- **Save/restore window (size and) position** — When selected, the position of each Data Monitor window is saved for use the next time you start Data Monitor. When selected, the position of your Data Monitor toolbars and window panels are saved for use the next time you start Data Monitor. If you have docked your toolbars and rearranged window panels, and want Data Monitor to use the new positions the next time you start up, select this feature. This makes it easy for you to pick up where you left off when you start a new session; all your windows are restored.

Save/restore toolbar positions (or properties) — When selected, the default values entered in the Preference dialog box for the open **Plot View** (6. *Plot View*, p.63) and **Plot XY View** (8. *Plot XY View*, p.67), are saved or restored.

The parameters in the **Writeback** panel (in a Windows host) are:

Don't ask before writeback — When selected, a warning message, usually displayed each time a value is written, is not displayed in the GUI.

The parameters in the **Applications Settings** panel (in a UNIX host) are:

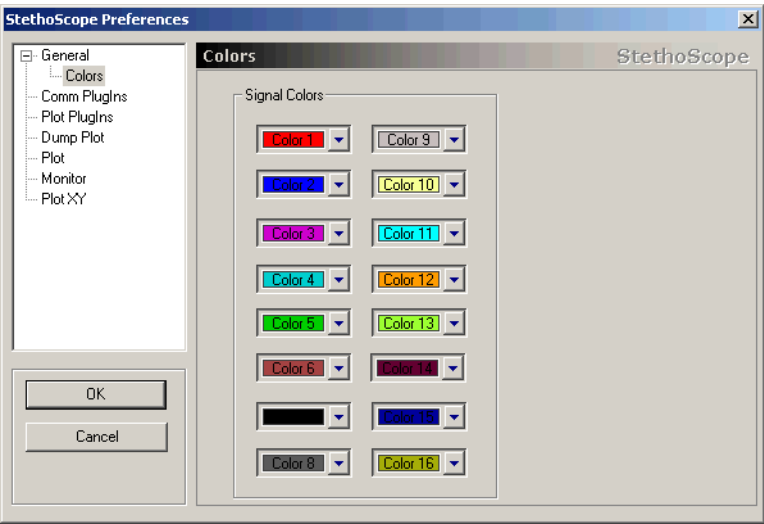
Change Application Font — Select a new font style from the menu.

Click **OK** to apply changes. The settings take affect immediately for the current session and all subsequent sessions, until changed again. Or click **Cancel** to exit the dialog box without saving your changes.

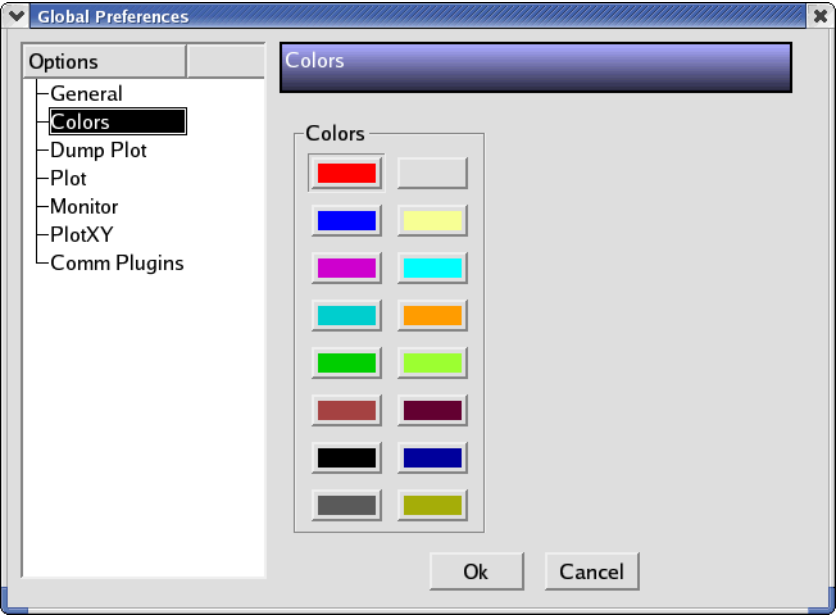
2. Colors View

The **Colors** view allows you to control the signal (trace line) color settings for the **Plot** and **Plot XY** windows, as described in Chapters [7. The Plot Window](#) and [8. The Plot XY Window](#) respectively.

Windows Host



UNIX Host

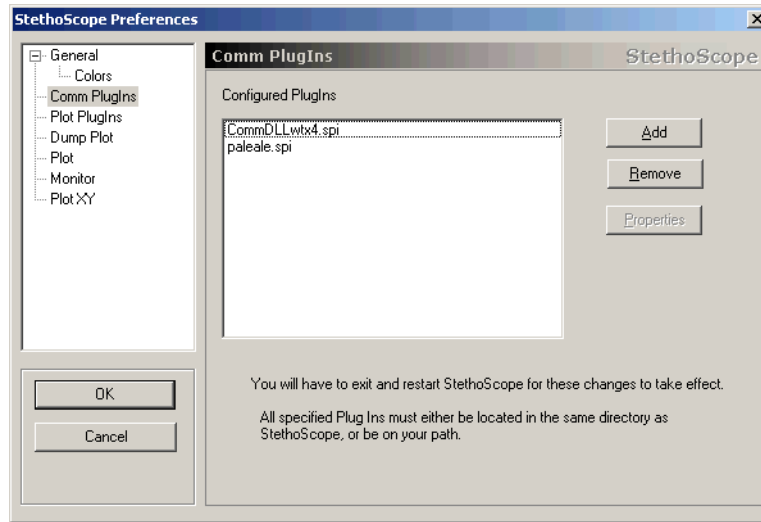


Color preferences take effect as soon as you click **OK**.

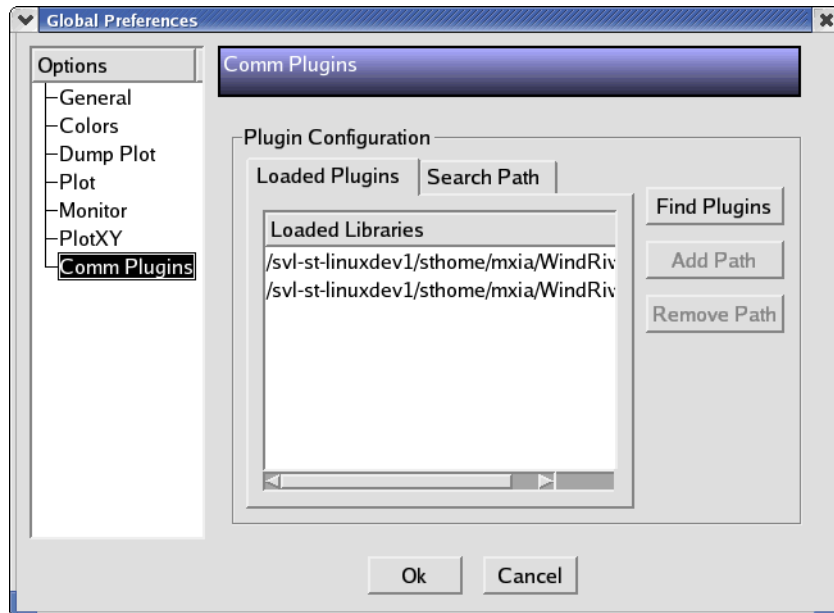
3. **Comm Plug-ins View**

The Comm Plug-ins view is used to specify the plug-ins you want to use for communications between the host GUI and the target.

Windows Host



UNIX Host



The **Configured Plugins (Plugin Configuration)** panel displays all the plug-ins currently configured. To delete a plug-in, first select it in this list, then click **Remove** to delete it.

The Windows host buttons are:

- **Add** — Click this button to bring up the **Open** dialog box in which you can navigate to, and select, plug-in files.
- **Remove** — Click this button to remove a plug-in selected in the Configured Plug-ins panel.
- **Properties** — Opens the **Properties** window where you can configure the selected plug-in.

The UNIX host controls are:

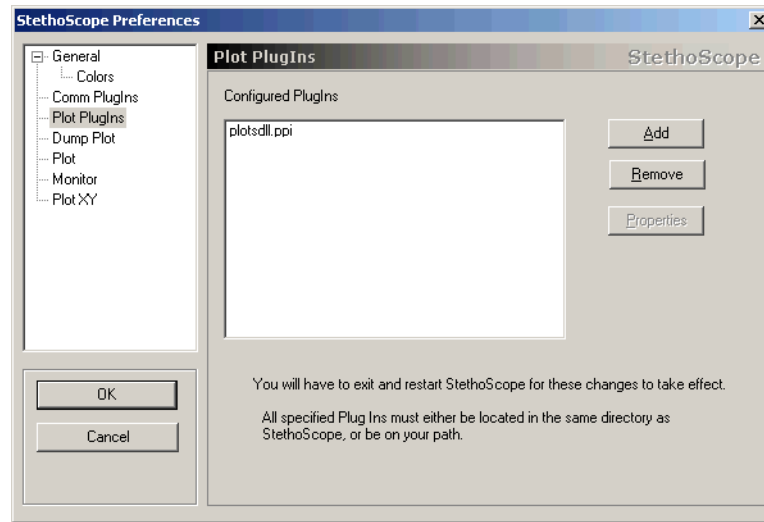
- **Find Plugins** — Opens the Find dialog box in which you can search for, and select, plug-in files. Use the **Search Path** tab view to view directory paths to search, and use **Add Path** or **Remove Path** buttons to this list.
- Configured plugins can be observed in the **Loaded Plugins** tab view.



NOTE: You must exit and restart Data Monitor to see the effects of modifying this list in either a Windows or UNIX host.

4. **Plot Plug-ins View** (Windows hosts only)

The Plot Plug-ins view is used to specify the plug-ins that control which plots are available for use in Data Monitor.



The **Configured Plug-ins** panel displays all the plug-ins currently configured. To delete a plug-in, first select it on this list, then click **Remove**.

The buttons are:

- **Add** — Click this button to bring up the **Open** dialog box in which you can navigate to and select plug-in files.
- **Remove** — Click this button to remove a plug-in selected in the Configured Plug-ins panel.
- **Properties** — Opens the **Properties** window where you can configure the selected plug-in.

Click **Ok** to apply the changes. The settings take effect immediately for the current session and all subsequent sessions. Click **Cancel** to exit the dialog box without saving your changes.

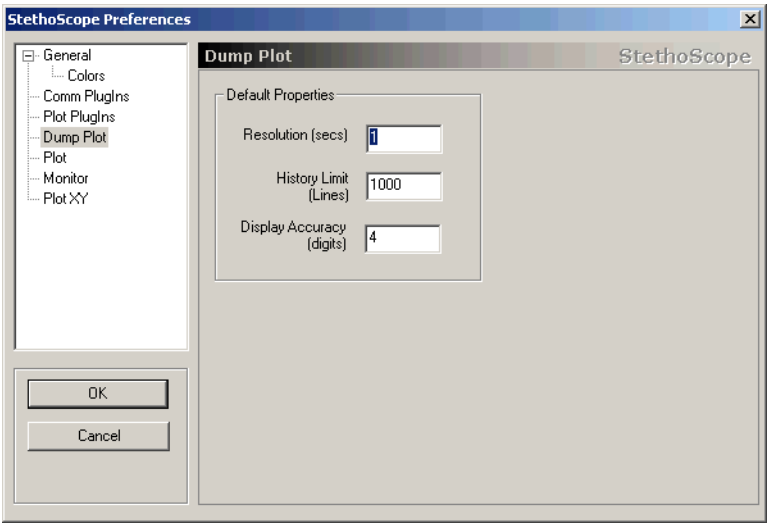


NOTE: You must exit and restart Data Monitor to see the effects of modifying this list.

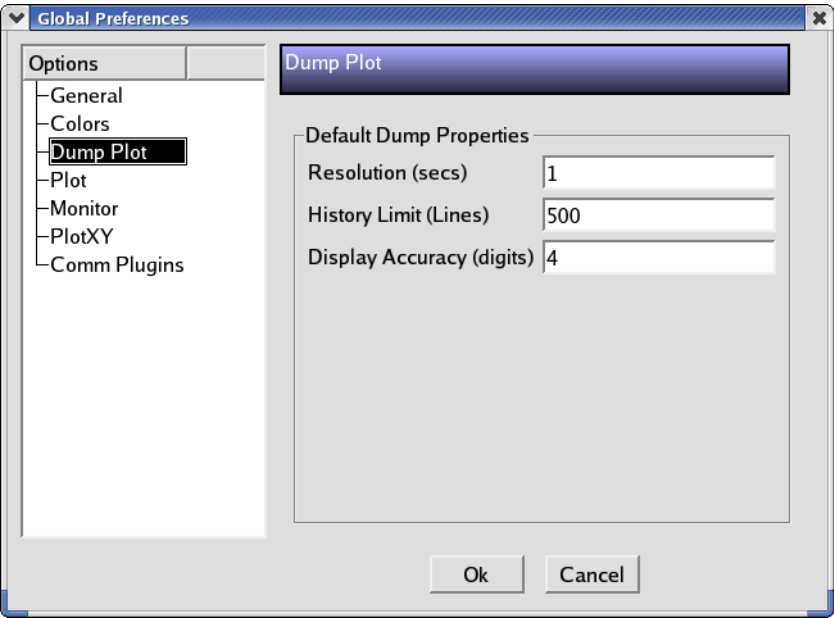
5. Dump Plot View

The Dump Plot view allows you to change the default values used when new Dump Plot data-display windows are created.

Windows Host



UNIX Host

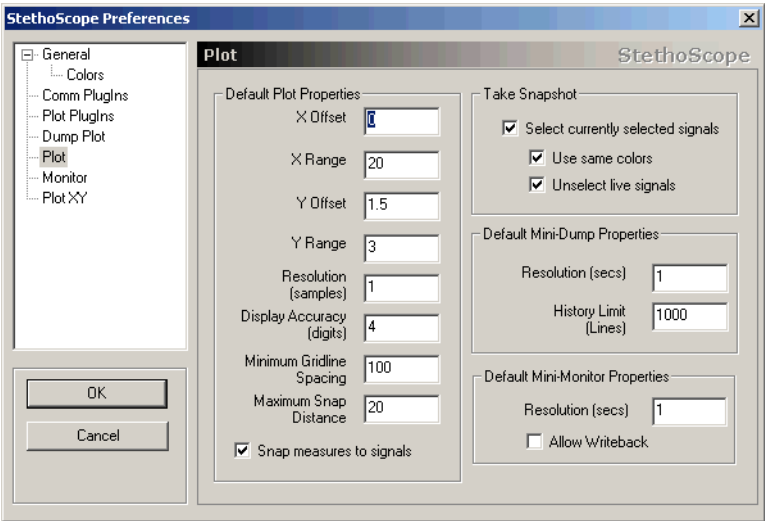


These preferences are described in detail in [9.3 Setting New Dump Plot Window Preferences](#), p.171.

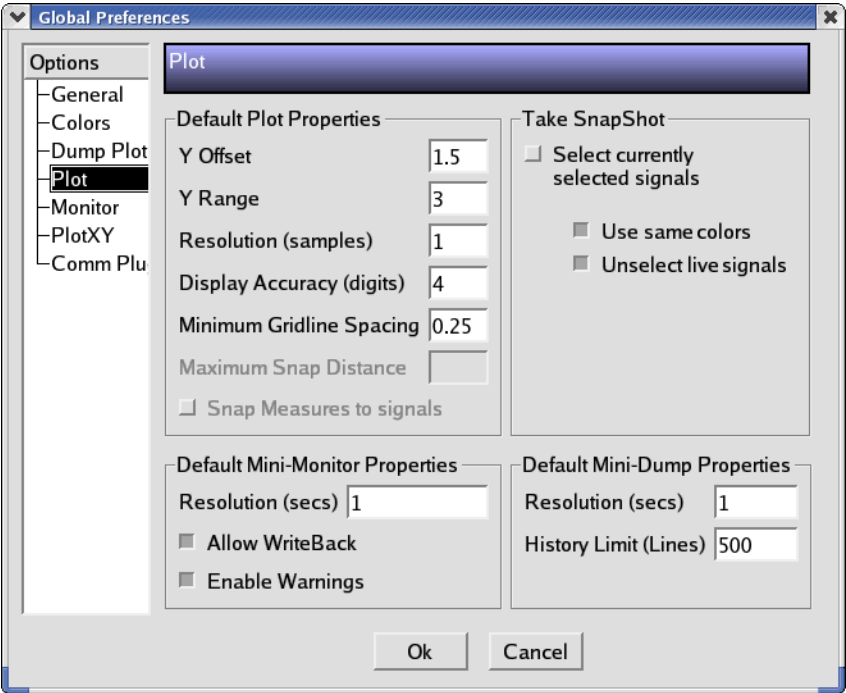
6. **Plot View**

The Plot view allows you to change the default values used when new Plot data-display windows are created.

Windows Host



UNIX Host

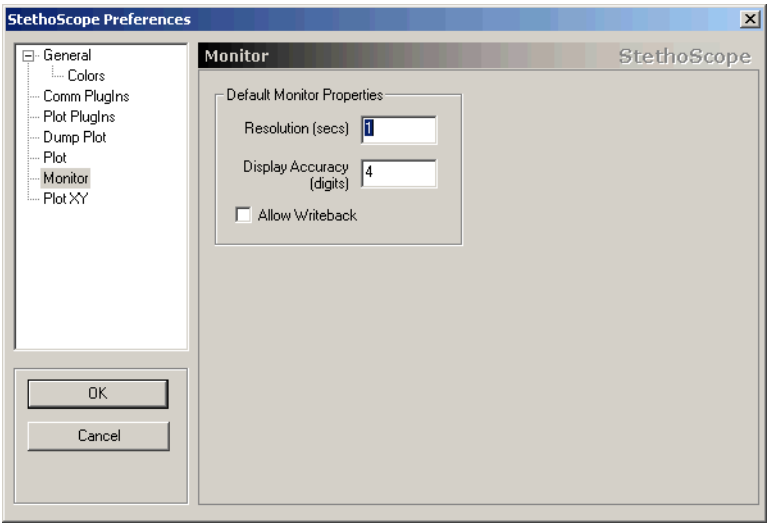


These preferences are described in detail in [7.6 Setting New Plot Window Preferences](#), p.134.

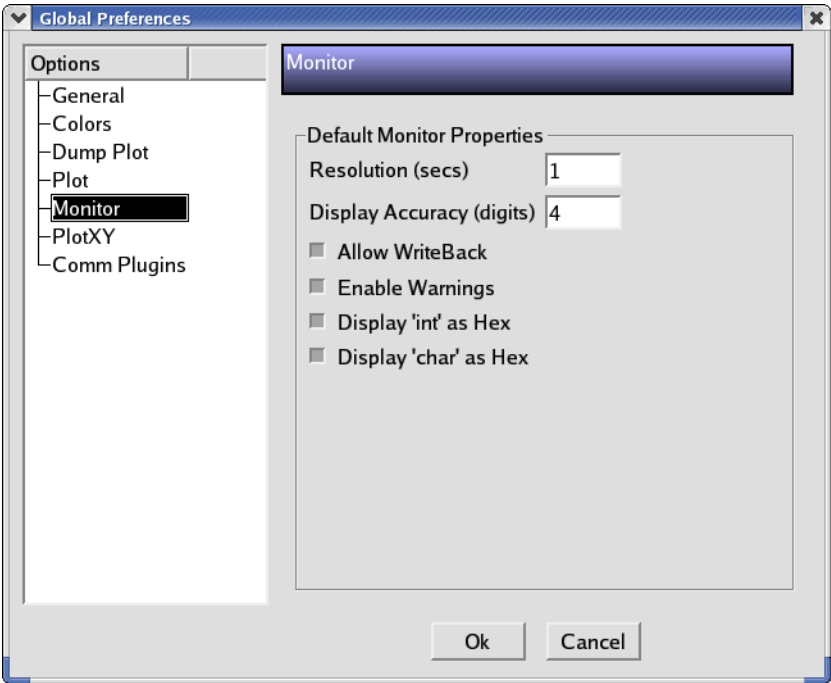
7. **Monitor View**

The Monitor view allows you to change the default values used when new Monitor data-display windows are created.

Windows Host



UNIX Host

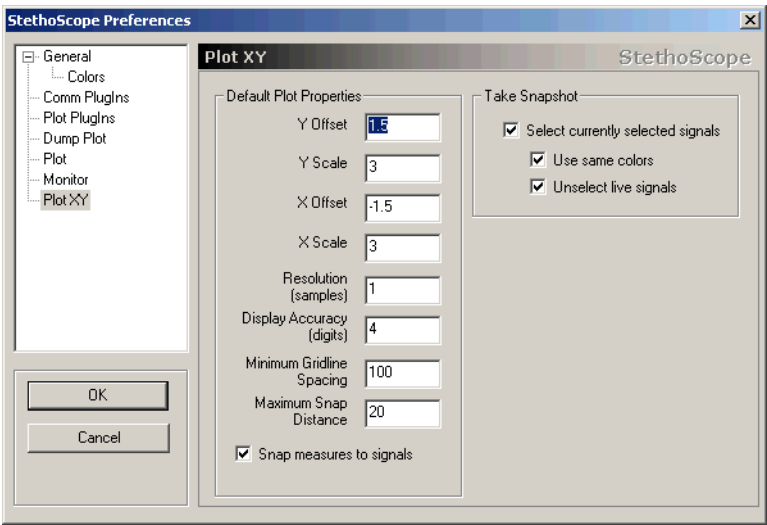


These preferences are described in detail in [10.4 Setting New Monitor Window Preferences](#), p.182.

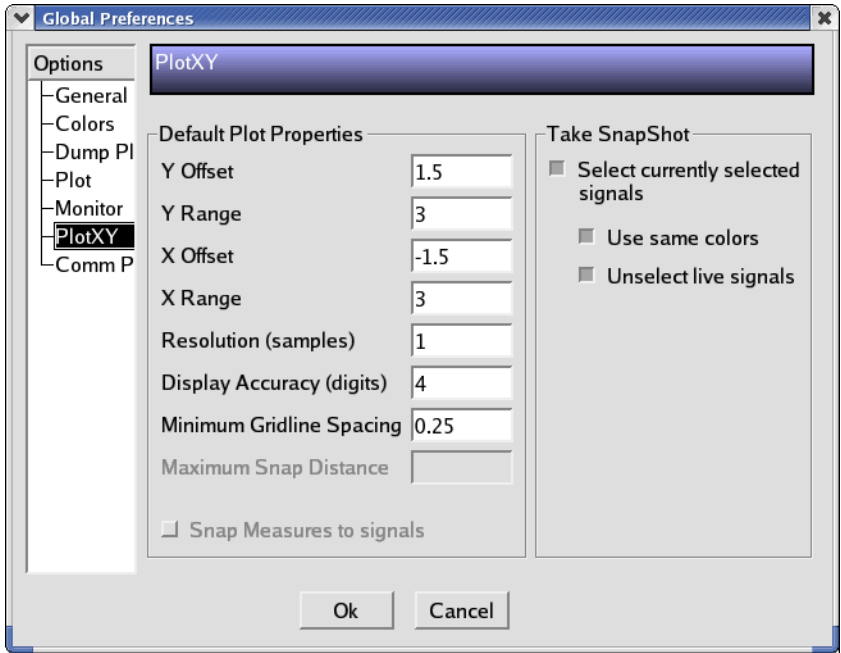
8. **Plot XY View**

The Plot XY view allows you to change the default values used when new Plot XY data-display windows are created.

Windows Host



UNIX Host



These preferences are described in detail in [8.5 Setting New Plot XY Window Preferences](#), p.160.

3.3.13 Close Window

Closes only the current window. Any remaining open Data Monitor windows are unaffected, except that if this is the only remaining open window, Data Monitor exits.

3.3.14 Exit Data Monitor

Quits the Data Monitor GUI, but does not stop target daemons. All Data Monitor windows are closed.

The secondary features and remaining functionality in the Data Monitor GUI is accessed through the remaining menu bar items. They are listed in the following sub-sections, but are described in detail in the various data-display window sections where they are applicable.

3.4 Menu Bar

The following items are available in the Data Monitor GUI menu bar.

Plot Menu Item (Windows Hosts Only)

In Windows hosts only, the Plot menu item contains commands for working with the contents of the **Plot** ([7. The Plot Window](#)), **Plot XY** ([8. The Plot XY Window](#)), **Dump Plot** ([9. The Dump Plot Window](#)), and **Monitor** ([10. The Monitor Window](#)) windows. It is unique in each of these chapters, where it is described in detail.

Note that there is no Plot menu item in a UNIX host.

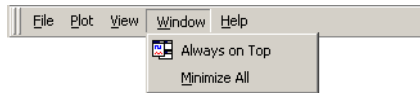
View Menu Item

The View menu contains commands for working with the contents of the **Plot** (7. *The Plot Window*), **Plot XY** (8. *The Plot XY Window*), **Dump Plot** (9. *The Dump Plot Window*), and **Monitor** (10. *The Monitor Window*) windows.

Note that commands in the **View** menu are not all available in all data-display window types. These menu commands, as well as some that are available in the various pop-up menus, are described in the chapters on each of the specific data-display windows (see Chapters 7, 8, 9, and 10).

Window Menu Item (Windows Hosts Only)

In a Windows host only, the **Window** menu contains items that affect the characteristics of currently open Data Monitor windows.



The Window menu item contains the following commands:

- **Always on Top**
Keeps the currently selected window as the topmost display on the desktop.
- **Minimize All**
Minimizes all Data Monitor windows.



NOTE: This menu item is not available in UNIX hosts.

Help

The Help menu currently has only one topic:

- **About Data Monitor**
Opens the **About** box displaying version and copyright information.

3.5 Pop-up Menus

Pop-up menus, available in Plot and Plot XY windows only, contain options that are unique to specific signals, and in some cases, to the cursor's location within the Data Monitor GUI.

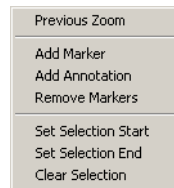
The following pop-up menus are described:

- **On-Grid**
- **On-Trace** (Windows hosts only)
- **Signals Tree**
- **Legend**

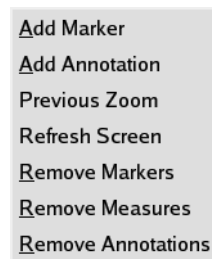
On-Grid

Several plot-related commands are available in the pop-up menu that appears when you right-click while passing the cursor anywhere over the grid area of a Plot window, except when directly over a plot line (see [7.2 Plot Window Tour](#), p.108).

Windows Host



UNIX Host



The following menu items are available in both Windows and UNIX hosts:

- **Previous Zoom**
Reverses the effect of the previous zoom action (see [Zooming](#), p.76). Note that the **Zoom to Fit** command erases the history of all previous zooms.
- **Add Marker**
Adds text to a specific point on the grid area, showing the coordinates of a desired point (see [Markers](#), p.76).

- **Add Annotation**

Adds text to a specific point on the grid area, you can type any comment you want (see [Annotations](#), p.77).

- **Remove Markers**

Deletes all markers and measures from the grid area (see [Markers](#), p.76).

Windows hosts only selections:

- **Set Selection Start**

Marks the beginning of an area of the graph to select for copying to the clipboard. This mark, a vertical line, is placed at the location of the mouse cursor at the time this option is selected. The selection is highlighted as soon as you select **Set Selection End** from the pop-up menu.

- **Set Selection End**

Marks the end of the graph area you want to copy to the clipboard. It also is placed at the location of the mouse cursor at the time this option was selected. The graph area between **Set Selection Start** and **Set Selection End** is immediately highlighted and ready to be copied into the clipboard using either the **Plot > Copy** menu command, or the standard **Ctrl+C** keyboard shortcut. The clipboard can then be copied to most text or bitmap handling applications. The results will vary between text lists or bitmap renderings depending on the application.



NOTE: You can also select the entire graph area thus far using the **Plot > Select All** menu command.

- **Clear Selection**

Deselects the graph area selected using the Set Selection Start and Set Selection End commands. Note that this does not delete the selected graph data itself.

UNIX host only selections

- **Refresh Screen**

Refreshes the screen data with the most current data.

- **Remove Measures**

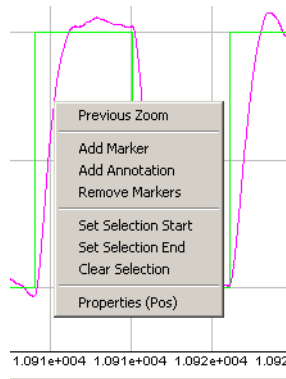
Deletes all measures from the grid area (see [On-grid Measurements](#), p.78).

- **Remove Annotations (UNIX only)**

Deletes all annotations from the grid area (see [Annotations](#), p.77).

On-Trace (Windows Hosts Only)

In the **Plot** window only, the pop-up menu that opens when you right-click exactly on a trace line is the same as the pop-up menu in *On-Grid*, p.71, but there is one additional menu item.



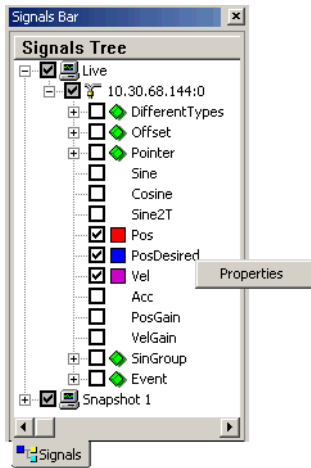
The additional menu item is:

- **Properties (Pos)**

Opens the **Signal Properties** dialog box for the Plot window, which is described in detail in *7.3 Signal Properties Dialog Box*, p. 123, and for the Plot XY window, in *8.4 Signal Properties Dialog Box*, p. 154. In this dialog box you can configure parameters that affect the appearance and behavior of the specific signal your cursor is on.

Signals Tree

A pop-up menu opens when you right-click a signal name in the **Signals** tab view (**Active Signals** tab view in a UNIX host) in the Signals Tree.



This menu has only one option:

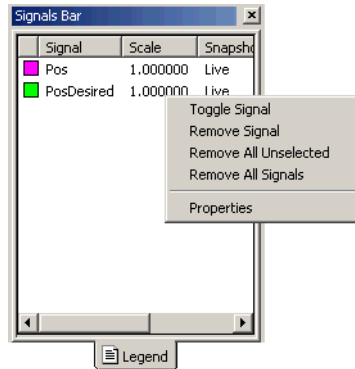
- **Properties (Signal Properties in a UNIX Host)**

Opens the **Signal Properties** dialog box, which, for the Plot window, is described in detail in [7.3 Signal Properties Dialog Box](#), p.123, and for the Plot XY window, in [8.4 Signal Properties Dialog Box](#), p.154. In this dialog box you can configure parameters that affect the appearance and behavior of the specific signal your cursor is on.

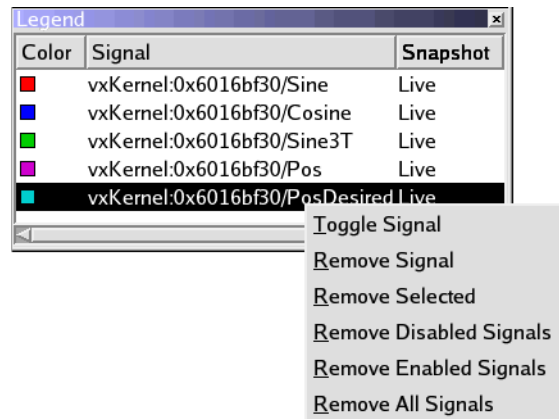
Legend

A pop-up menu opens when you right-click a signal name in the **Legend** tab view of a **Plot** or **Plot XY** window (or in the **Legend Window** in a UNIX host, see [Legend Window \(UNIX Hosts Only\)](#), p.27).

Windows Host



UNIX Host



The pop-up menu has the following options.

- **Toggle Signal**
Turns the selected signal **on** or **off**. This is the equivalent of going to the **Signals Tree** and alternately checking and deselecting the signal check box, except that it leaves the signal name in the Legend list when you toggle it **off**.
- **Remove Signal**
Removes only the selected signal from the graph (regardless of its toggled status).
- **Remove All Unselected**
Removes all signals that are toggled **off**.
- **Remove All Signals**
Removes all signals (regardless of their toggled status).

Windows hosts only selections:

- **Properties**
Opens the **Signal Properties** dialog box. For the Plot window, it is described in detail in [7.3 Signal Properties Dialog Box](#), p. 123, and for the Plot XY window, in [8.4 Signal Properties Dialog Box](#), p. 154.

UNIX hosts only selections:

- **Remove Disabled Signals**

Removes all the signals that have been toggled off.

- **Remove Enabled Signals**

Removes all signals that are currently toggled on (displayed).

3.6 Screen Operations

You can enhance the data graphing area with options, including zooming in and out, placing markers at strategic points on the graph, adding text annotations at any place on the graph, panning and moving the graph area, and taking and displaying on-grid measurements. These operations are described in the following sections. They are shown in the on-grid pop-up menu in the figure above.

Zooming

You can magnify a region to see details by zooming. The offset and scale of the plot are adjusted so that the zoomed region fills the Plot window.

To zoom in to a desired region, do the following:

1. Press and hold the **Shift** key.
2. Click and drag with the left mouse button to select a region of the plot.

The on-grid pop-up menu includes a **Previous Zoom** command, which can be used to return to the previous zoom. Note, however, that using the **Zoom to Fit** menu command (or the **Zoom to Fit** toolbar button - see [Plot Window Toolbar](#), p.41) erases the history of all zoom actions.

Markers

A marker shows the coordinates of a point on the grid.

To add a marker, do one of the following:

- Right-click at the desired point in the grid area. Select **Add Marker** in the pop-up menu,

or,

- Press and hold **Ctrl** while clicking the mouse.

To delete a specific marker, do the following:

- Drag the marker off the grid area.

To delete all markers and measures, do one of the following:

- Right-click anywhere in the grid area, then select **Remove Markers** in the pop-up menu,

or,

- Select the **Plot > Delete Markers** menu command.

Annotations

An annotation is text that you type in to mark a point on the grid.

To add an annotation, do the following:

- Right-click at the desired place in the grid area, then select **Add Annotation** in the pop-up menu.

To move an annotation, do the following:

- Drag the annotation to the desired spot on the grid area.

To delete a specific annotation, do one of the following:

- Drag the annotation off the grid area,
or (**UNIX** only),
▪ Right-click the annotation, then select **Delete Annotation** from the **On-Grid** pop-up menu.

Panning

Panning provides an easy way to change the offset of a plot by allowing you to move the plot directly from within the display window. This is useful especially when you have zoomed in and you want to see an adjacent region.

After zooming in, you can pan in the X and Y directions. When not zoomed in, you can only pan in the Y direction since the entire X range is already in view.

To pan the view region, do one of the following:

- Click and drag with the left mouse button to move the viewing region,
or,
- Use the scroll bars.

On-grid Measurements

You can create a line on the grid that measures the distance between any two points. As you move the mouse pointer over the plot area, the coordinates of the mouse pointer are displayed in the rightmost panel of the status bar at the bottom of the window.

To measure offsets between any two points on the plot, do the following:

1. Press and hold the **Ctrl** key.
2. Place the mouse pointer at the first location on the grid.
3. Click and drag the mouse pointer to the second location.
4. Release the mouse button and the **Ctrl** key.

This creates a line between the two points and displays the difference between the two points as an x, y pair.

To delete a measurement, do one of the following:

- Click an end-point of a measurement and drag the end-point off the plot window,
or,
- Select **Plot > Delete Markers** to remove all markers and measurements.

3.7 Status Bar

The status bar is located at the bottom of the Data Monitor **Plot** window.



It is hidden or displayed with the **View > Status Bar** menu command, and is divided into three panels:

- The left panel provides a description of the command when the mouse pointer is passed over any Plot window toolbar button.
- The middle panel displays status messages. Double-click this panel at any time to display the Log window and see a history of status messages.
- The right panel displays the XY coordinates of your mouse pointer when it is passed over the plot grid.

4

Using the Signal Manager

4.1 Introduction 81

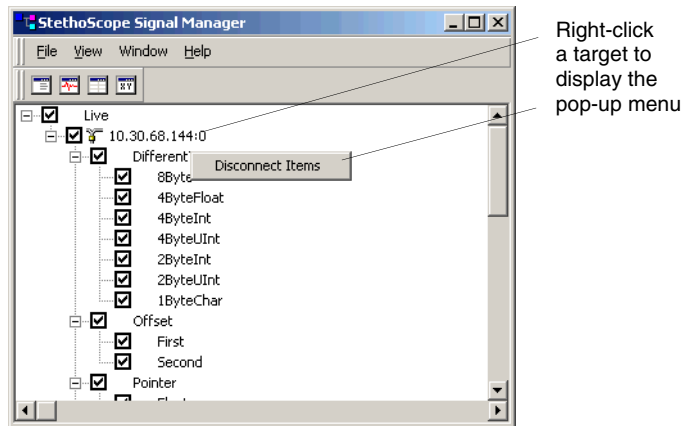
4.2 Using the Signal Manager Window 81

4.1 Introduction

A signal from your running target program can be displayed in the Data Monitor GUI only if it has been **installed** either before or during the Data Monitor session. The **Signal Manager** utility, available in the Data Monitor GUI, will do this for you. This chapter gives you the information you need to install all the variables you want to look at in the various Data Monitor windows.

4.2 Using the Signal Manager Window

Open the Signal Manager window with the **File > Signal Manager** menu command (or the **Signal Manager** toolbar button - see [Main Toolbar](#), p.40), or by right-clicking on a target in the **Signals Tree** and selecting **Install Signal** in the pop-up menu.



This window allows you to determine which signals are collected from each target. Only these signals, when installed, can be monitored in any of the four types of data-display windows.

Working With Signal Trees

Installed signals are displayed in a tree structure in the **Signal Manager** window, as well as in the **Signals Bar** panel of the individual data-display windows described in [2.4 The Data Monitor GUI](#), p.19. Use this Signal Manager window to choose which signals are active (collected from the target). In the data-display windows (**Plot**, **Plot XY**, **Dump Plot**, and **Monitor**), you use the **Signals Tree** in the Signals Bar to select the signals to display in the graph or table in the window.

To activate a signal (so that it is collected from the target), simply click a signal entry in the Signals Tree in the Signal Manager window. A check mark appears in the check box to show that it is active. To stop collecting a signal, click it again to clear the check mark.

Signals must be installed before they appear in the Signals Tree. (You can install signals using the **Signal Installation** dialog box; see [15. Installing Signals](#)). Signals Trees in other windows are identical, except that they only show signals activated in the **Signal Manager** window.

The Signals tree has the following characteristics:

- Signals may be organized hierarchically when they are installed. A directory entry—indicated by a "+" or "-" node icon—contains sub-entries that are either signals or other directories.
- A directory entry always begins a new branch on the signals tree. You can expand or collapse these branches by clicking the "+" or "-" icon, respectively.
- Each entry (signal or directory) has a check box. A check box in a directory indicates what is selected underneath it. A check box in a signal indicates whether or not that signal is selected. Check boxes indicate the following:
 - ☐ = No signal in this branch is selected, or, if a signal, this signal is not selected.
 - ☒ = All signals in this branch are selected, or, if a signal, this signal is selected.
 - ☒ = At a branch node, some, but not all, signals in this branch are selected.
- Selecting the check box at a directory node in a Signals Tree selects everything underneath it. Similarly, clearing (deselecting) a directory check box clears everything underneath it.
- Click a signal or branch check box to toggle its selection—click an unselected signal to select it; click it again to deselect it.
- Right-clicking a target in a Signals Tree displays a pop-up menu with the following commands:

Disconnect Items

Disconnects the Data Monitor GUI from the selected live target (see [Disconnecting the Target](#), p.84).



NOTE: If you see an empty Signals Tree, either you are not running an application instrumented with the Data Monitor API on the target, your target program did not install any signals, or you have only registered but not activated any signals (You can observe this last case by opening the Signal Manager window and seeing that signals are listed but none are checked).

The best way to become familiar with the signals tree is to experiment while running the demo program. You will see how easy it is to make signals active in the Signal Manager window, and to select or clear signals in the data-display windows.

Installing Signals

A signal name is the name you assign to a data item that is collected from your target application by Data Monitor. An installed signal is one that has been **registered** and **activated**, as described in [Installing Signals](#), p.250.

Signals can be installed in any of 3 ways:

- One at a time, automatically by variable name or manually by address, using the **Signal Installation** dialog box, opened with the **Install signal to Data Monitor** button on the Workbench toolbar, where you can configure all the necessary parameters (see [15.2 Using the Signal Installation Dialog Box](#), p.232).
- Instrumenting your VxWorks target code using the Data Monitor API (see [A. API Reference: VxWorks](#)), or Linux target code using the Data Monitor API (see [B. API Reference: Linux](#)).
- Using an executable file that is itself fully instrumented with the Data Monitor API (**samplertask.so** for VxWorks, or **processsampler** for Linux), and which you can run on your target (see [15.4 Code Instrumentation Alternative](#), p.241).

Disconnecting the Target

Right-click a target name in the Signal Manager window to display a pop-up menu and select **Disconnect Items**. This action severs the communication link between the Data Monitor GUI and the target. This does not, however, stop the collection thread or Data Monitor daemons on the target. You may reconnect the GUI to the same, or connect to a different, scope index using the **New Target Connection** dialog (see [3.3.1 Connect to Target](#), p.45).

5

Triggering

5.1	Introduction	85
5.2	Configuring a Trigger	85
5.3	Setting a Trigger	93

5.1 Introduction

As described earlier (see [Triggering](#), p.50), the Data Monitor API module on the target is responsible for handling periodically collected data (or samples) and sporadically collected data (or events). The **Triggering** facility in Data Monitor provides control over when and how often these samples are collected. This chapter describes the Triggering facility and how it is used.

5.2 Configuring a Trigger

The Data Monitor triggering facility supports a single trigger at any given time. Triggering is disabled when the Triggering dialog box is not open, or it is open but

the trigger has not yet been armed, or has been disabled. In this state, calls to **ScopeCollectSignals()** result in data being collected normally.

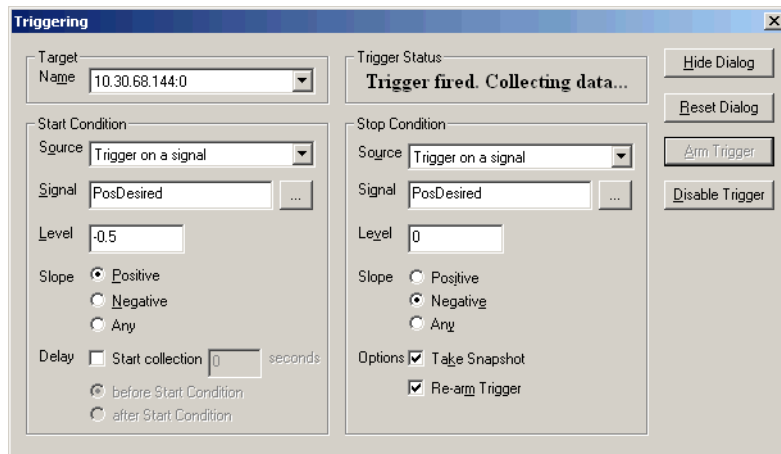
When the Triggering dialog box is open and the trigger is armed with valid start and stop conditions, all calls to **ScopeCollectSignals()** from that point on return without collecting any data. In this armed state (that is the trigger is valid, but has not yet fired), sampled signals are not plotted in any of the GUI windows, although **event** data will continue to be plotted as before. Collection and plotting of samples begins when the **Start Condition** is met (that is when the trigger fires). Data continues to be plotted on the GUI until the **Stop Condition** occurs. At that point the trigger is then either disabled or rearmed depending on the **Re-arm** option specified in the Triggering dialog box.

Triggering Dialog Box - Windows Host

The Triggering option is set up and executed in similar ways in both a Windows and UNIX host. However, there are sufficient discrepancies in the details to warrant describing them separately.

In a Windows host, open the Triggering dialog box using the **File > Triggering** menu command (or the **Triggering** toolbar button - see [Main Toolbar](#), p.40).

Windows Host



The control settings in the Triggering dialog box in a Windows host are grouped in the following descriptive panels:

- **Target**
- **Start Condition**
- **Trigger Status**
- **Stop Condition**

Target

In a Windows host, the Target field is a drop-down menu displaying a list of target names or TCP/IP addresses attached to the Data Monitor GUI. Select the target on which you want to configure triggering.

Start Condition

These are the settings that start the trigger. The trigger is fired and data collection begins when the trigger conditions you specified in the Start Condition panel are met.

The following triggering Start Condition parameters can be set:

- **Source**
Specifies the specific kind of triggering source for the Start Condition. Select one of the following from the drop-down list:
 - **Trigger on a signal**—specifies the source to be a periodically sampled signal.
 - **Trigger on an event**—specifies the source to be an event.
 - **Trigger immediately**—causes triggering to start immediately without waiting on a condition. If this option is chosen, there is no need to fill out the other parameters for Start Condition.
- **Signal**
If you selected **Trigger on a signal**, any non-derived signal may be used as the trigger source. This excludes deactivated signals (those not being collected and displayed by Data Monitor). Choose a signal using the Browse "... button.
If you selected **Trigger on an event**, type in the **event ID**.
- **Level**
Enter the signal value which should set off the trigger. This is relevant only if you selected **Trigger on a signal**.

- **Slope**

If you selected **Trigger on a signal**, specify the slope for the trigger:

- **Positive**—the trigger fires only if the source signal value is **increasing** when it crosses the trigger level.
- **Negative**—the trigger fires only if the source signal value is **decreasing** when it crosses the trigger level.
- **Any**—the trigger fires when the source signal value crosses the trigger level without respect to direction.
- **Delay**—if checked, it allows you to delay the trigger firing for the time (in seconds) you specified in the **Start Collection** field.
- **Before Start Condition**—if checked, the delay begins immediately, before the **Start Condition** begins to be evaluated.
- **After Start Condition**—if checked, the delay begins immediately after the Start Condition has been met.

Trigger Status

This field displays the current status of the Data Monitor trigger.

Stop Condition

Triggering is disabled when the trigger conditions you specified in the Stop Condition panel are met. Note that data collection continues even after triggering is disabled, unless **Re-arm** is selected for the trigger.

The following trigger Stop Condition parameters can be set.

- **Source**

Specifies the specific kind of triggering source for the Stop Condition. Select one of the following from the drop-down list.

- **Trigger on a signal**—specifies the source to be a periodically sampled signal.
- **Trigger on an event**—specifies the source to be an event.
- **Trigger after set time period**—This causes triggering to terminate after the number of seconds specified in the **Secs** text field has elapsed.

- **Signal**

If you select:

- **Trigger on a signal**—any non-derived signal may be used as the trigger stop source. This excludes deactivated signals (those not being collected and displayed by Data Monitor). Choose a signal using the Browse "..."
button.
- **Trigger on an event**—type in the **eventID**.
- **Level or Secs**
This field label changes to **Secs** if you selected **Trigger after set time period** in **Source** above; otherwise the label is **Level**. If **Level**, enter the signal value which should stop the trigger. If **Secs**, enter the number of seconds of triggering after which you want triggering to stop.
- **Slope**
If you selected **Trigger on a signal**, specify the slope for the trigger:
 - **Positive**—the trigger stops data collection only when the source signal value is increasing when it crosses the trigger level.
 - **Negative**—the trigger stops data collection only when the source signal value is decreasing when it crosses the trigger level.
 - **Any**—the trigger stops when the source signal value crosses the trigger level without respect to direction.
- **Options**
Optional actions to take after the trigger stops:
 - **Take Snapshot**—if this box is checked, a snapshot is taken of the signal and events that were collected between the start and stop conditions.
 - **Re-arm Trigger**—if this option is chosen, the trigger is automatically rearmed after the Stop Condition is met. This is useful in cases where you might want to observe an occurrence that happens over and over again. Choosing the **Take Snapshot** option along with this option can help you monitor occurrences during an unattended overnight run.

Buttons

The following buttons cause the trigger, with its options set, to take the indicated actions:

- **Hide Dialog**
Hides the dialog box. The trigger is only active while the dialog box is open. If you want to cancel the action of the trigger, click this button.

- **Reset Dialog**

Resets all options to their default values.

- **Arm Trigger**

Causes the calls to **ScopeCollectSignals()** from that point on to respond to the conditions set for this trigger, until you either **Disable Trigger** or **Hide Dialog** (that is, the trigger will now fire when the conditions are met).

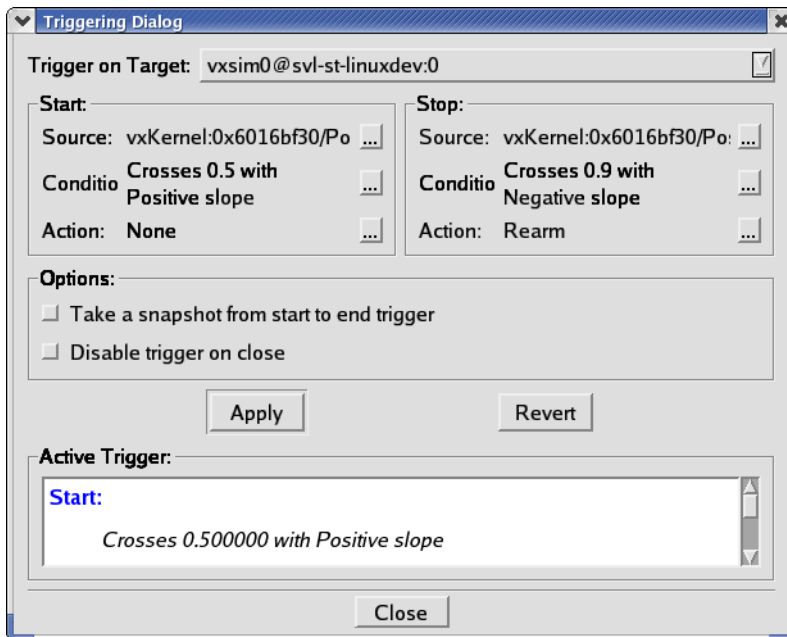
- **Disable Trigger**

Causes the trigger to become disarmed without closing the dialog box (configured values remain set). Data collection resumes as it was before the trigger was armed.

Triggering Dialog Box - UNIX Host

In a UNIX host, open the Triggering dialog box using the **File > Triggering** menu command (or the **Triggering** toolbar button - see [Main Toolbar](#), p.40).

UNIX Host



The control settings in the Triggering dialog box in a UNIX host are grouped in the following descriptive panels:

- **Trigger on Target**
- **Start**
- **Stop**
- **Options**
- **Active Trigger**

Trigger on Target

The Trigger on Target field is a drop-down menu displaying a list of target names and index numbers attached to the Data Monitor GUI. Select the target on which you want to configure triggering.

Start

These are the settings for starting the trigger. The trigger is fired and data collection begins when the trigger conditions you specified in the Start condition panel are met.

In the Start condition panel, the following triggering parameters are set using the "..." button accompanying each parameter. The options are:

- **Source**
Select the signal to be the initial trigger source.
- **Condition**
Select the condition to trigger the action.
- **Action**
Select the action to be taken when the triggering condition specified above has been met.

Stop

Triggering is disabled when the trigger conditions you specified in the Stop condition panel are met. Note that data collection continues even after triggering is disabled, unless trigger **Rearm** is selected.

The following trigger stop parameters can be set using the "..." button accompanying each parameter:

- **Source**

Select the signal to be the terminating trigger source. It may be different from the Source signal in the Start panel.

- **Condition**

Select the condition to disable triggering.

- **Action**

Select the action to be taken when the stop condition specified above has been met.

Options

These are optional actions you can specify to be taken after the trigger is stopped.

- **Take a snapshot from start to end trigger**

Causes a snapshot to be taken of the signal and events that were collected between the start and stop conditions.

- **Disable trigger on close**

Causes the trigger to become disarmed when the dialog box is closed. Data collection resumes as it was before the trigger was armed.

Active Trigger

This field displays the status of the Data Monitor trigger.

Buttons

The following buttons cause the trigger, with its options set, to take the indicated actions:

- **Apply**

Causes the currently selected parameters to take effect, but leaves the dialog box open for any additional changes.

- **Revert**

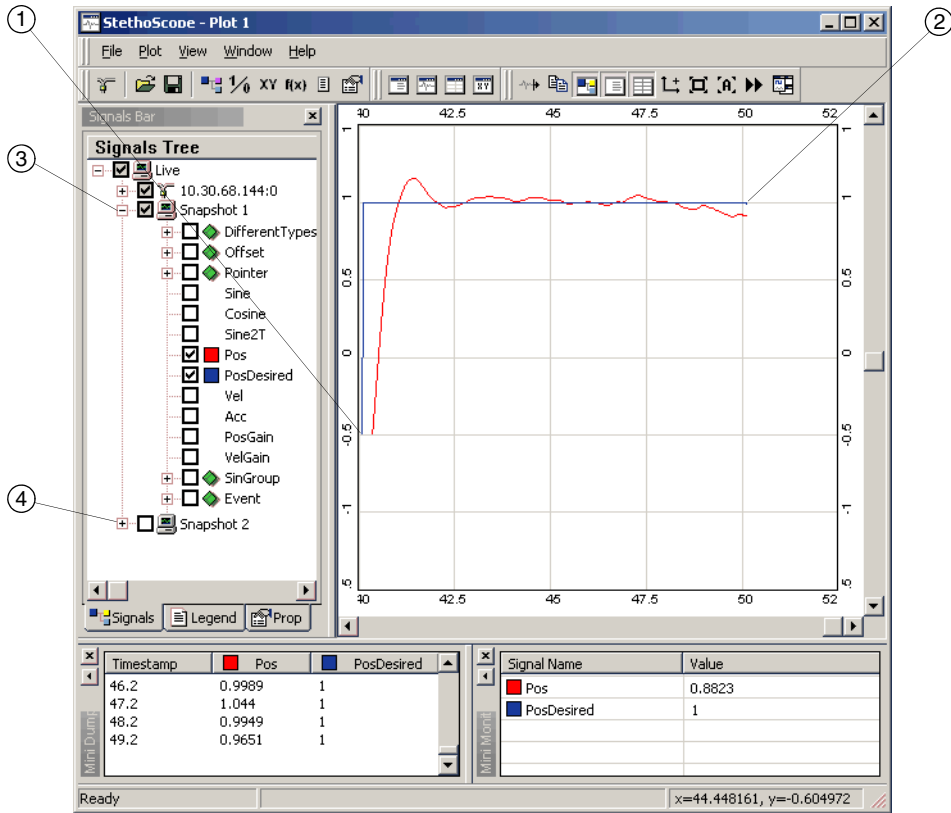
Reverts to the parameter settings in effect before the last Apply action was taken.

- **Close**

Closes the dialog box.

5.3 Setting a Trigger

This section walks you through the process of setting an example trigger. We will use the Data Monitor demonstration program introduced in [2. Getting Started](#) for this example. The example output from this demonstration is shown and annotated below. Refer to it in the text that follows.



The Chain of Events

In the GUI display above you can follow the numbered steps as Data Monitor responds to the trigger parameters set up in the example dialog box shown in [Triggering Dialog Box - Windows Host](#), p.86.

1. The trigger fires when the value of PosDesired crosses **0.5** on a positive slope.
2. The trigger is disabled as soon as PosDesired starts on a negative slope.
3. A snapshot of the collected data is created and listed in the Signals Tree, as well as being displayed in the graphing area.
4. Data Monitor continues to rearm the trigger and create more snapshots.

Trying it Yourself

You can set up and run the same sample trigger shown in [Triggering Dialog Box - Windows Host](#), p.86 by following these steps:

- Step 1:** Start the demonstration program (see [2.5 Testing Your Installation](#), p.30).
- Step 2:** Connect the Data Monitor GUI to the scope index of the demo program.
- Step 3:** For viewing, select the Pos and PosDesired signals from the Signals Bar.
- Step 4:** Open the Triggering dialog box and follow the steps below for Windows or UNIX host:

Windows Host

- a. In the **Target** panel, choose the target from the **Name** drop-down menu.
- b. In the **Start Condition** panel, choose **Trigger on a signal** for **Source**.
- c. Select **PosDesired** for the **Signal**.
- d. Set **Level** to **-0.5**.
- e. Select **Positive** for **Slope**.
- f. In the **Stop Condition** panel, choose **Trigger on a signal** for **Source**.
- g. Set **PosDesired** for the **Signal** (same as for the **Start Condition**).
- h. Set **Level** to **0**.
- i. Select **Negative** for **Slope**.

Finally, there are optional actions to take after the trigger is stopped:

- j. Select **Take Snapshot**.
- k. Select **Re-arm Trigger**.

For the trigger to take effect, click **Arm Trigger**.

UNIX Host

- a. Choose the target from the **Trigger on Target** drop-down menu.
- b. In the **Start** panel, select a **Source** signal using the "..." button; the **Pos** signal is used in this example.
- c. For **Condition**, select **Crosses 0.5 with Positive slope** (using the "..." button, and entering the value 0.5).
- d. For **Action**, select **None**.
- e. In the **Stop** panel, select **Pos** again for Source.
- f. For **Condition**, select **Crosses 0.9 with Negative slope** (using the "..." button, and entering the value 0.9).
- g. For **Action**, select **Rearm**.

Finally, there are optional actions to take after the trigger is stopped:

- h. Select **Take a snapshot from start to end trigger**.
- i. Select **Disable trigger on close**.

The trigger you set using these commands causes periodic collection to be temporarily disabled until the Start Condition is met.



NOTE: No signals are plotted during this wait period in either host GUI.

Step 5: With dialog box open, wait for the Start Condition to be met and data to be displayed.

Signal plotting resumes when the value of the **PosDesired** signal crosses 0.5 with a positive slope (**Step 1** in the figure above). Data collection and plotting will continue for about 8 seconds before the **Stop Condition** is satisfied (**Step 2**). At the end of this period, a snapshot of the data collected during those eight seconds is taken and added to the **Signals Tree** in the **Signals Bar** under the name of the target (**Step 3**). Check the other signal names in the snapshot to view their data during this time period.

If **Re-arm Trigger** was selected in a Windows host (or **Rearm** was selected for **Action** in the **Stop** panel in a UNIX host), the trigger is automatically rearmed and this causes the above scenario be repeated forever (**Step 4**).

Step 6: Click Disable Trigger

This causes Data Monitor stop triggering at any time. You can do the same thing by just closing the dialog box.

Step 7: Choose some signals from the live data view again

The selected signals have become the snapshot signals since they began appearing, and you should observe data being plotted as usual.

6

Derived Signals

6.1 Introduction 97

6.2 Creating Derived Signals 98

6.1 Introduction

A **Derived Signal** is a signal whose value is computed by mathematical operations on other signals. Derived signals are calculated by Wind River Data Monitor on the host, from live signals. They are not sampled directly from your real-time system. The derived signals facility provides a simple means of scaling and offsetting signals, plotting differences and ratios of signals, and so forth.

The following are some examples of derived signals.

```
PosError    = PosDesired - Pos
Pos10       = Pos * 10
Pos10Plus3  = Pos10 + 3
Tan         = Sin / Cos
LogError    = log10 PosError
```

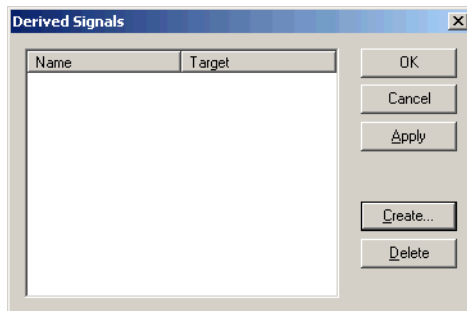
This chapter describes derived signals in detail, including how they are created and how to troubleshoot them.

6.2 Creating Derived Signals

You must create a derived signal using the **Derived Signals** dialog box before it will appear in any of the Data Monitor data-display windows. To create a derived signal, follow these steps.

Step 1: Open the Derived Signals dialog box.

Do this using the **File > Derived Signals** menu command (or the **Derived Signals** toolbar button - see [Main Toolbar](#), p.40).



Step 2: In the Derived Signals dialog box, click Create.

The first in a series of **Derived Signal Wizard** dialog boxes opens.



Step 3: Enter a name for the new signal in the Name text box.

Make sure the name is unique and preferably descriptive.

Step 4: Use the Type drop-down menu to select a type for the derived signal.

For instance, a 4-byte integer, char, and so forth.

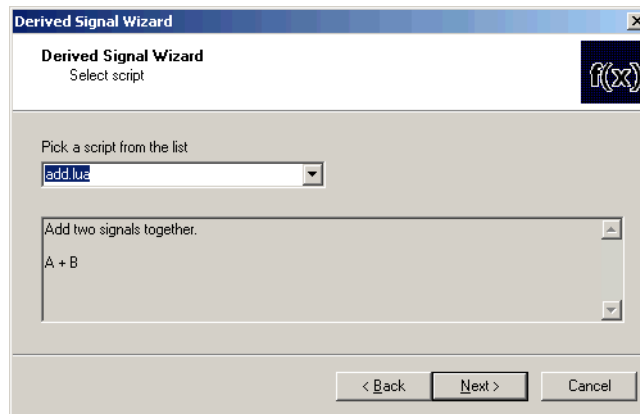
Step 5: Use the Target drop-down menu to select one of your connected targets.



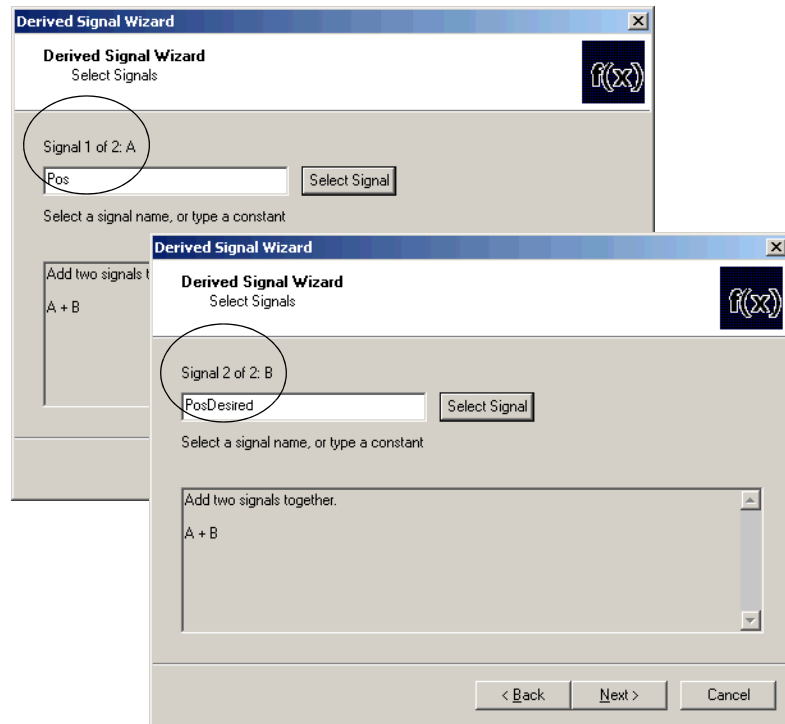
NOTE: The dialog box fields and instructions for creating a derived signal diverge between Windows and UNIX host requirements at this point. They converge again at Step 6.

If you are running in a **Windows** host, do the following steps:

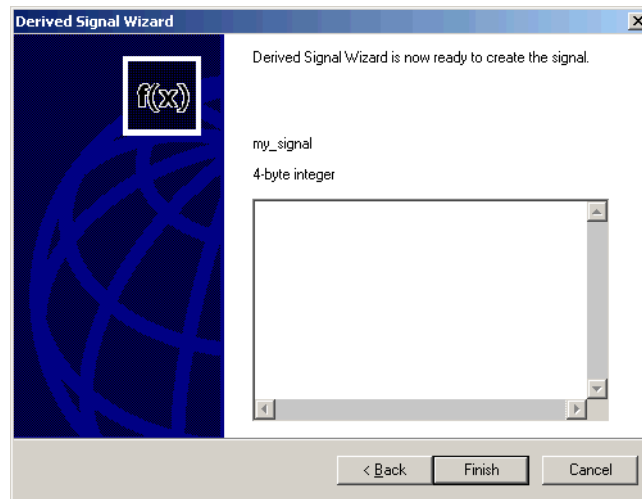
1. Click **Next** to open the next **Derived Signal Wizard** dialog box.



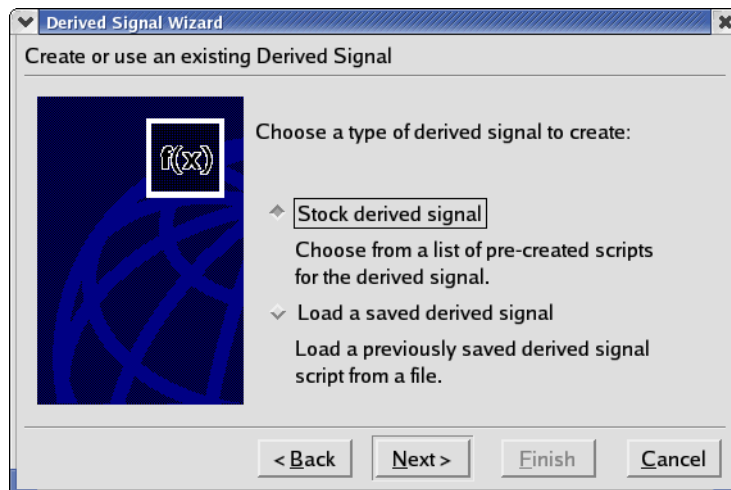
2. Use the drop-down menu in the **Pick a script from the list** field to select a mathematical operation to use in creating your derived signal (you can see a list of the script items in [Table 6-1](#)).
3. Click **Next** to open the next **Derived Signal Wizard** dialog box.



4. Use this dialog box to select signal(s) to apply to the script you previously selected. If the script you selected requires two or more signals (as in this example), this dialog box asks you to enter the first signal, and the dialog boxes following ask for the second and subsequent signals until they have all been selected. Note that it prompts you for which signal you are about to enter (circled).
5. Click **Next** to open the final **Derived Signal Wizard** dialog box.

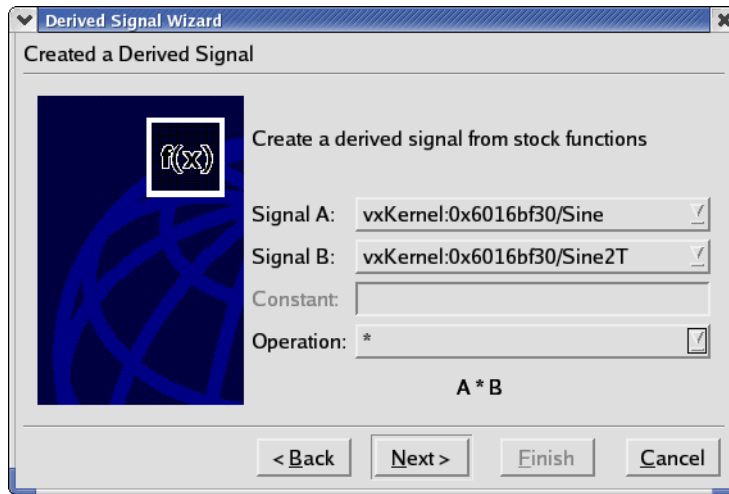


6. In this dialog box you can inspect the completed derived signal.
If you are running in a **UNIX** host, then do these steps:
1. Click **Next** to open the next **Derived Signal Wizard** dialog box.

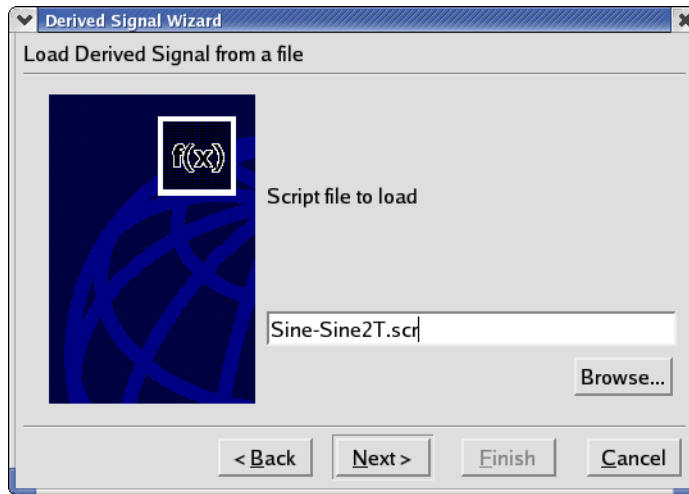


In this dialog box, you can choose whether you want to use a pre-defined (stock) script to create your derived signal, or load one of your own previously derived (saved) signal scripts from a file.

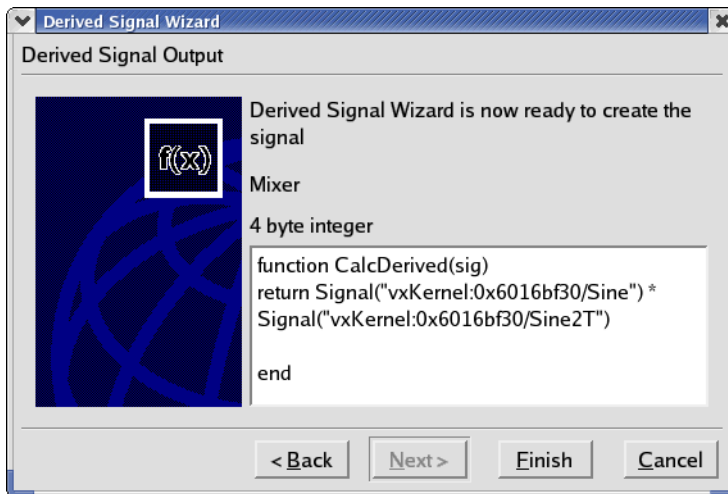
2. Choose **Stock derived signal** for this exercise.
3. Click **Next** to open the next **Derived Signal Wizard** dialog box.



4. Select one or more signals (as appropriate) to apply to the script you selected in the previous step.
5. Click **Next** to open the next **Derived Signal Wizard** dialog box.



6. In this dialog box, specify the pathname and file containing the saved script you want to use.
7. Click **Next** to open the final **Derived Signal Wizard** dialog box.



8. In this dialog box you can inspect the completed derived signal.

Step 6: In either a Windows or UNIX host, click Finish to create and save the derived signal.

You can also click **Back** to make any desired changes before finishing, or click **Cancel** to exit without saving the new signal

When you click **Finish**, you will arrive back at the **Derived Signals** dialog box, which displays the newly created **Derived Signal** in its list. Click **Apply** to save your new signal if you want the dialog box to remain open to create more **Derived Signals**, or Click **OK** to save the new signal and exit the dialog box. Or you can click **Cancel** to close the dialog box without saving the newly created **Derived Signal**.

When computing derived-signal values, Data Monitor truncates any infinite values to a value of **+/-10,000,000.0**. This prevents numeric problems with recursively defined signals.

You can modify a derived signal in a UNIX host using the **Edit** button and change it by hand; however, it is recommended that you delete the existing signal and re-enter your changed values. In a Windows host, you cannot edit an existing derived signal; you can only delete it and create another one with your changed values.

To delete a derived signal, select the signal from the list in the **Derived Signals** dialog box, then click **Delete** (or use the **Delete** key on the keyboard).

Mathematical Operations

Mathematical operations you can use to create derived signals are shown in [Table 6-1](#).

Table 6-1 **Derived Signal Operations**

Operation	Meaning
abs	abs(A)
add	Add two signals together (A + B)
arccos	if abs(A)<=1 then arccos(A), else arccos(sign(A)), result in degrees
arcsin	if abs(A)<=1 then arcsin(A), else arcsin(sign(A)), result in degrees
arctan	atan(A), result in degrees
arctan2	if (B==0) then 10000000*sign(A), else atan2(A,B)
avg	Average signal value

Table 6-1 **Derived Signal Operations** (cont'd)

Operation	Meaning
cos	cos(A), where A is in degrees
divide	if (B==0) then 10000000*sign(A), else A/B
exp	exp(A)
exp10	exp10
filter1	A * previous_filter1 + (1-A)*B, where initial_filter1 = B
hypot	sqrt(A*A + B*B)
ln	ln(A)
ln10	ln10(A)
mult	Multiply two signals together (A * B)
saturate	if (A>B) then B, if (A<-B) then -B, else A
sin	sin(A), where A is in degrees
sqrt	sqrt(abs(A))
sub	Subtract two signals (A - B)
tan	tan(A), where A is in degrees
add	abs(A)

Troubleshooting Derived Signals

When a **Derived Signal** is defined, it should appear in the **Signals Tree** of every **Plot**, **Dump Plot**, and **Monitor** window.

There are two reasons derived signals may not appear:

- The derived signal has been toggled off in the **Signal Manager**.
- The signal will appear only when all of its operands are defined. If any of the operand signals do not appear in the buffer being viewed (perhaps caused by a typing mistake), then the derived signal will not be available in that buffer.

7

The Plot Window

- 7.1 Introduction 107
- 7.2 Plot Window Tour 108
- 7.3 Signal Properties Dialog Box 123
- 7.4 Axis Properties Dialog Box (Windows Hosts Only) 128
- 7.5 Displaying Events 130
- 7.6 Setting New Plot Window Preferences 134

7.1 Introduction

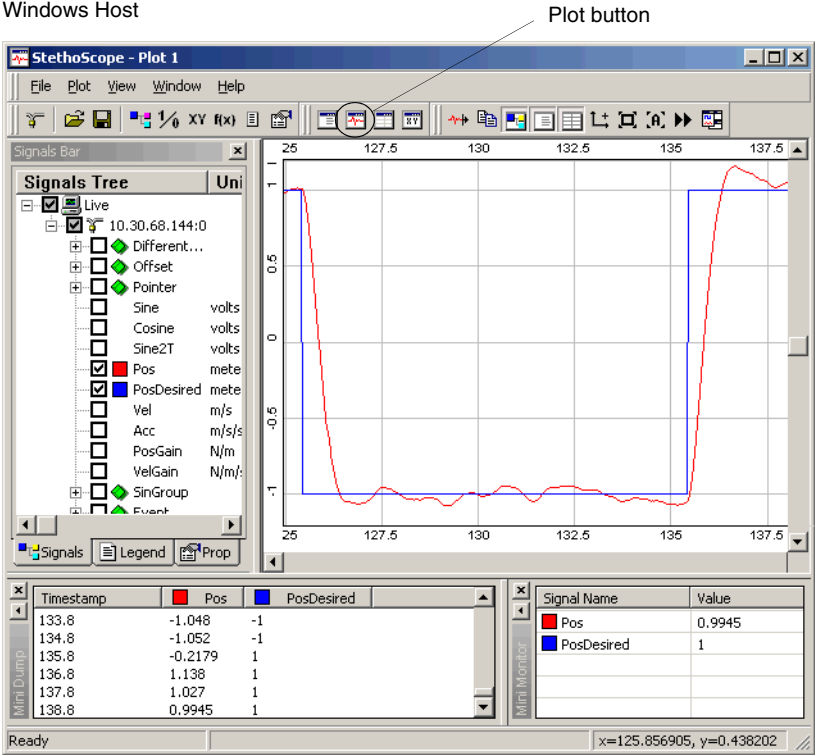
The **Data Monitor Plot** window graphically displays real-time signals plotted over time. Each signal appears on the plot grid in a different color, with a legend showing the color each signal is plotted with. This chapter describes the **Plot** window in all its detail, and shows you how to use its features.

Before using a **Plot** window, it may help to understand how and when data is collected from the target—please refer to [Notes and Hints](#), p.258.

7.2 Plot Window Tour

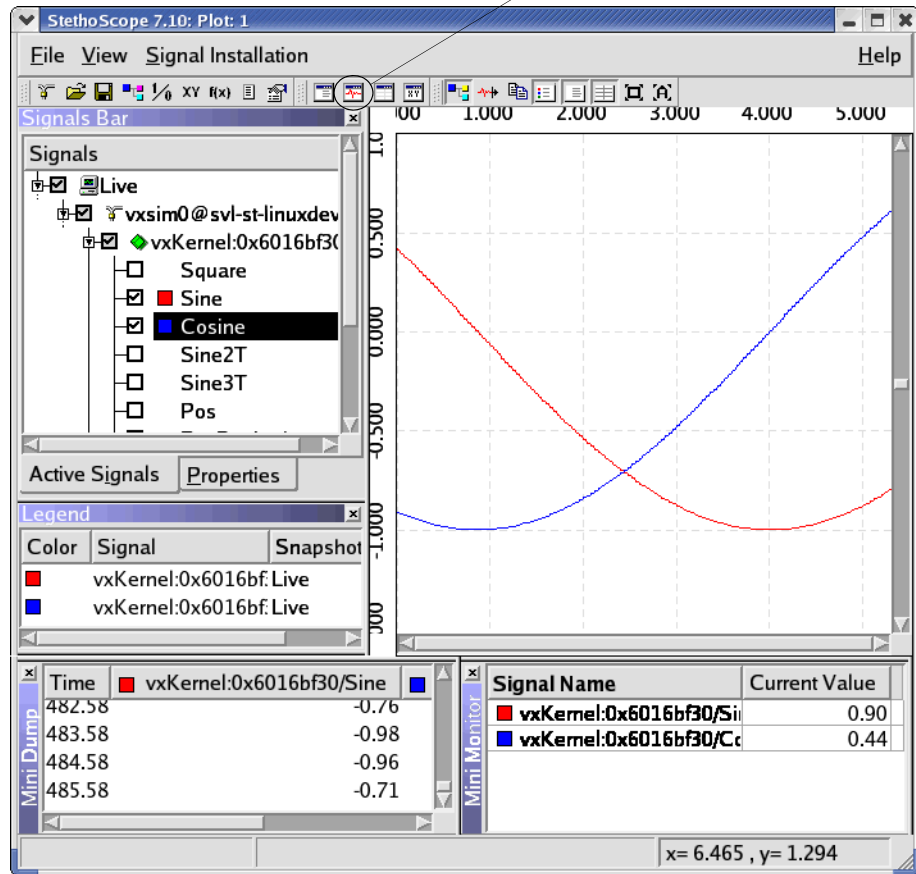
The **Plot** window, an example of which is shown here in both a Windows and a UNIX host, opens when you first start Data Monitor. Open additional Plot windows using the **File > Plots** menu command (or the **Plot** toolbar button - see [Main Toolbar](#), p.40).

Windows Host



UNIX Host

Plot button



Selecting Signals

To select signals for display, follow these steps:

1. If you do not already have a **Signals Bar** open in your **Plot** window, open one using the **View > Signals Bar** menu command (or the **Signals Bar** button). Make sure the **Signals** tab is selected so that a **Signals Tree** is displayed.
2. Use the Signals Tree to select which signals you want to display in the graph. The signals in this tree were installed using the **Signal Manager** (see [4. Using](#)

the Signal Manager). Selecting a signal from the Signals Tree causes the following actions:

- The signal data is plotted in the window using the next available plot color. You can select a different color using the **Signal Properties** dialog box (see [7.3 Signal Properties Dialog Box](#), p.123), or the **Preferences** dialog box (see [2.Colors View](#), p.56).
- The **Legend** tab view is updated to include the selected signal. The Legend shows you what color is being used for each selected signal. You can also use the Legend to select and clear signals. The Legend tab view is described in [Legend Tab View \(Windows Hosts Only\)](#), p.117.

Most of the remaining functionality in the **Plot** window is provided through the toolbars and menus. These items are described in detail in the following sections:

- [3.3 File Menu Item](#), p.43
- [3.2 Toolbars](#), p.40
- [3.5 Pop-up Menus](#), p.71

Popup Menu

When you right-click anywhere in the plot area, the **On-grid** pop-up menu opens with several options pertaining specifically to the plot area. These additional options are described in detail in [3.5 Pop-up Menus](#), p.71.

Screen Operations

The screen operations outlined for the **Plot** window also apply to the **Plot XY** window. These operations are described in [3. Data Monitor Features](#), and include:

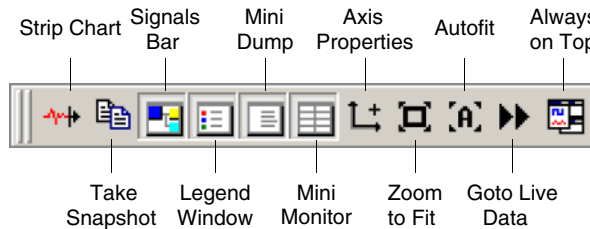
- [Zooming](#), p.76
- [Markers](#), p.76
- [Annotations](#), p.77
- [Panning](#), p.77
- [On-grid Measurements](#), p.78

Toolbar

In a default **Plot** window, the first two toolbars are identical to the toolbars shown in [Main Toolbar](#), p.40 and [Plots Toolbar](#), p.41, with the menu items described here

also. But the right-most toolbar is specific to the **Plot** window. All the toolbars are dockable (as well as the menu bar), which means you can move them to other locations, on or off the window, simply by dragging them. Each of the toolbars can be independently displayed or hidden using the **View** menu item (see [View Menu Item](#), p.113).

This **Plot** window toolbar is a subset of the one described in detail in [Plot Window Toolbar](#), p.41.

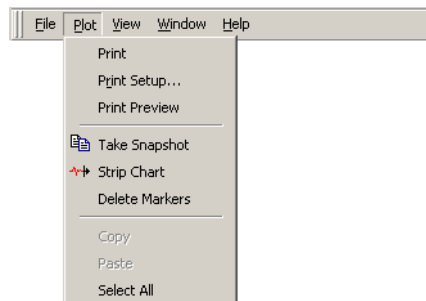


Menu Bar

Some of the **Menu Bar** items are discussed in detail in [3.2 Toolbars](#), p.40, where it is mentioned that the **File**, **Plot** (in Windows hosts only) and **Help** menu items (as well as the **Window** menu item in a Windows host) are consistently the same across all data-display windows. The **Plot** (in Windows hosts only) and **View** menu items, however, contain some commands that are unique to the **Plot** and **View** windows. These menu items are described in detail in the following section.

Plot Menu Item (Windows Hosts Only)

In a Windows host, the **Plot** menu item contains commands for working with the data displayed in the **Plot** window.



In a Windows host only, the **Plot** menu commands for the **Plot** window are:

- **Print**
Prints the signal traces in the data-display window. It opens a **Print** dialog box where you can configure your printer options.
- **Print Setup**
Allows you to select printer parameters and characteristics before printing.
- **Print Preview**
Lets you see, on a print mock-up screen, what the printed page will look like before actually printing it.
- **Take Snapshot**
Saves a copy of all the active signals (not just the selected signals) for all connected targets. For more information, see [11.2 Utilizing Snapshots](#), p.185.
- **Strip Chart**
Changes the display so that it presents a continuous, scrolling plot (instead of repainting the plot when it fills every 10 seconds). Any snapshots you may be displaying on the grid when you select **Strip Chart** become unselected; that is, snapshots do not appear on the grid while **Strip Chart** is selected. This command is only available in the **Plot** window. For more information, see [Strip Chart](#), p.122.
- **Delete Markers**
Deletes all markers, annotations, and measures from the grid area. For more information, see [Markers](#), p.76, [Annotations](#), p.77, and [On-grid Measurements](#), p.78 respectively.
- **Copy**
Copies the selected content to the clipboard, and deselects the selected data. Note that this option is greyed out and unavailable unless you have some data in the plot area selected.
- **Paste**
Copy the contents of the clipboard to the window you are in, at the insertion point. Note that this options is greyed out and unavailable until you have copied some data to the clipboard.

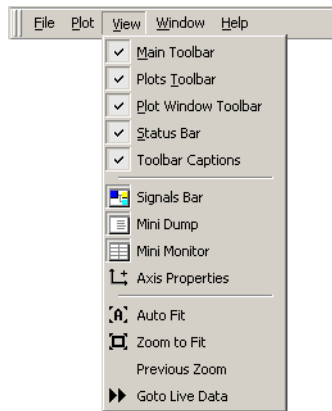
- **Select All**

Selects and highlights the entire plotted area up to the instant you selected the option, enabling you to copy it to the clipboard.

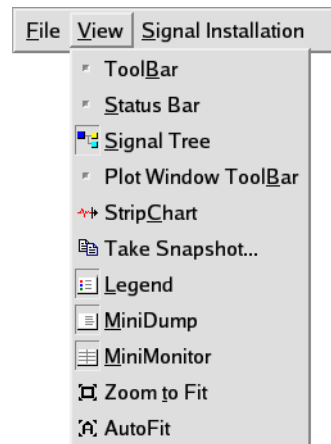
View Menu Item

The **View** menu item, in both a Windows and a UNIX host, contains commands for displaying toolbars and auxiliary views in the **Plot** window, as well as some **Plot** window-specific commands.

Windows Host



UNIX Host



The following menu options are available in both Windows and UNIX hosts:

- **Plot Window Toolbar**

Controls whether the toolbar used within a specific data-display window is visible. The buttons represent items from the **Plot** and **View** menus for the specified data-display window. For more information, see [Toolbar](#), p.110.

- **Status Bar**

Controls whether the status line along the bottom of the window is visible. For more information, see [3.7 Status Bar](#), p.78.

- **Signals Bar**

Controls whether a **Signals Bar** panel appears in the window. The **Signals Bar** includes tabs for **Signals**, **Legends**, and **Properties**. For more information, see [Signals Bar](#), p.116.

- **Mini Dump**

Creates a **Mini Dump Plot** window within the **Plot** window. This is simply a smaller version of the **Dump Plot** window. It lets you see a running history of signal values. This command is only available in the **Plot** window. For more information, see [Mini-Dump Window](#), p.28.

- **Mini Monitor**

Creates a **Mini Monitor** window within the window. This is simply a smaller version of the **Monitor** window. It lets you peek at, and poke, data on the target. This command is only available in the **Plot** window. For more information, see [Mini-Monitor Window](#), p.28.

- **Auto Fit**

Causes the plotting area to be scaled dynamically and automatically to the extremes of the data in the Y direction being displayed in the window. It thus allows all data points to appear on the plot. It toggles on or off, but when on, it essentially has the effect of using the **Zoom to Fit** button continuously. This command is only available in the **Plot** and **Plot XY** windows.

- **Zoom to Fit**

Changes the scales and offsets so that all the signals fit and take up the entire **Plot** window. This option is only available in the **Plot** and **Plot XY** windows. See also **Previous Zoom**.

Additional options for **Windows** hosts only are:

- **Main Toolbar**

Controls whether the toolbar representing a selection of items from the **File** menu is displayed on Data Monitor's toolbar. For more information, see [3.2 Toolbars](#), p.40.

- **Plots Toolbar**

Controls whether the toolbar representing a selection of items from the **File > Plot** menu command is displayed on Data Monitor's toolbar. For more information, see [3.2 Toolbars](#), p.40.

- **Toolbar Captions**

Controls whether the names of the panels (**Signals Bar**, **Mini-Dump**, and **Mini-Monitor**) are displayed in the **Plot** window above their respective invocations.

- **Axis Properties**

Allows you to set parameters that control the scaling and placement of both grid lines and their plot values on the X and Y axis of a **Plot** or **Plot XY** plotting area. For more information, see [7.4 Axis Properties Dialog Box \(Windows Hosts Only\)](#), p.128.

- **Previous Zoom**

Reverses the action of the last zoom action. Note that when you use the **Zoom to Fit** command, the history of all previous zoom actions is lost, therefore this command has no effect immediately after a **Zoom to Fit** command. Also, if no previous zoom has been set, this item is greyed out and unavailable. This command is only available in the **Plot** and **Plot XY** windows.

- **Goto Live Data (Windows only)**

Returns to plotting live data when a snapshot is selected in the Signals Bar.

Additional options for UNIX hosts only are:

- **ToolBar**

Controls whether the toolbar representing a selection of items from the **File** menu is displayed on Data Monitor's toolbar. For more information, see [3.2 Toolbars](#), p.40.

- **Strip Chart**

Controls whether the Strip Chart representing a selection of items from the **File > Plot** menu command is displayed on Data Monitor's toolbar. For more information, see [Strip Chart](#), p.122.

- **Take Snapshot**

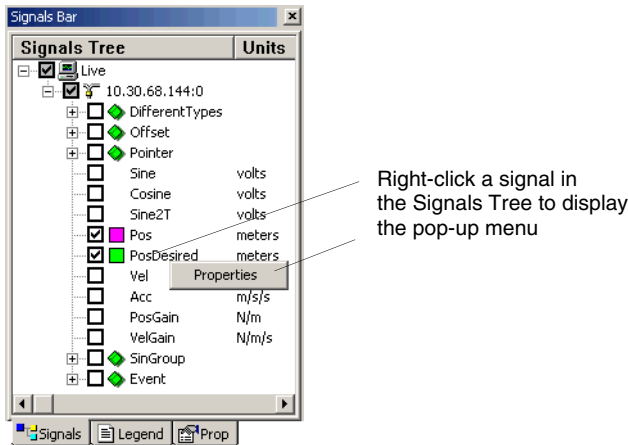
Saves a copy of all the active signals. For more information, see [11.2 Utilizing Snapshots](#), p.185.

- **Legend**

Controls whether the Legend window is displayed. For more information, see [Legend Window \(UNIX Hosts Only\)](#), p.27.

Signals Bar

The **Signals Bar** is a sub-window in the Data Monitor GUI that allows you to view and configure information that controls what is plotted in the graph area.



To open a **Signals Bar** if one is not already displayed, use the **View > Signals Bar** menu command (or the **Signals Bar** toolbar button - see [Plot Window Toolbar](#), p.41).

The **Signals Bar** contains the following tab views:

- **Signals Tab View**
- **Legend Tab View (Windows Hosts Only)**
- **Properties Tab View**

Signals Tab View

The **Signal Tree** in the **Signals** tab view (see the figure above) displays a tree structure containing the names of all the signals available to be plotted.

The installed signals are shown in an expandable tree structure containing signals that have been installed by the **Signal Manager** (see [4. Using the Signal Manager](#)). Traces on the graph are color-coded to the signals selected in the **Signals Tree**.

The Signals Tree is displayed by default when you open a **Plot** window, but it may be re-displayed at any time by opening the **Signals** tab view with the tab at the bottom of the sub-window.

The Signals Tree allows you to easily locate any signal that has been installed and add it to the graph of signal traces. Each node of the tree, and each signal, is a check box. Selecting the check box for an individual signal adds that signal to the graph, or, conversely, clearing the check box removes that signal from the graph. If you select a node check box, all the signal check boxes belonging to that node (but not including nodes below it) are selected at once and their signals are added to the graph. Conversely, clearing a node check box removes all signals below that node from the graph with the single click.

If a node check box is selected, but has a grey fill, it indicates that some, but not all, of the signals belonging to that node are selected and displayed on the graph. You may have to scroll down to see which ones are selected.

The Signals Tree is expanded or collapsed using the icon to the left of each node check box as follows:

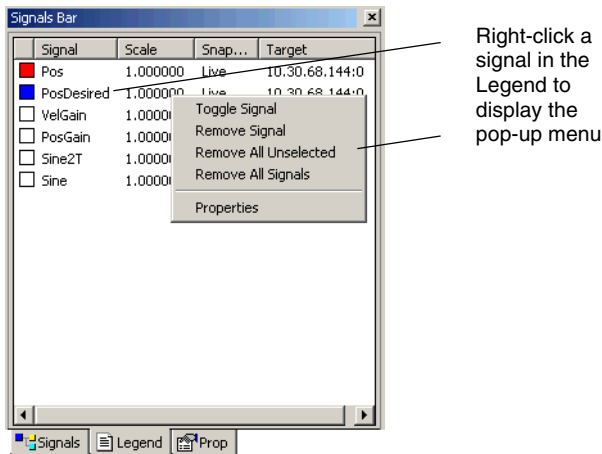
- "+" = **collapsed**; select to expand down to the next node(s).
- "-" = **expanded**; select to collapse up to this node.

The **Units** column in the **Signals** tab view shows the physical measurement units of each signal.

When you right-click a signal in the Signals tab view, a pop-up menu opens containing a single menu item as shown in the figure above.

Legend Tab View (Windows Hosts Only)

In a Windows host, the **Legend** tab view shows you what color is assigned to each signal on the plot. You can also use it to select which signals to plot.



NOTE: In a UNIX host, the **Legend** appears in a separate window, rather than a tab view. For details, see [Legend Window \(UNIX Hosts Only\)](#), p.27.

The columns in the **Legend** tab view display current option settings for each **Signal**, including:

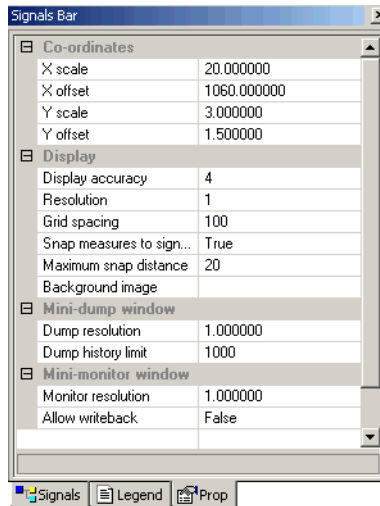
- **Scale**
The scale factor applied to second (Y) component signal values in relation to the first (X) component signal values.
- **Snapshot**
Whether the data is live, or from a snapshot.
- **Target**
The name and scope index of the target you are connected to.

When you right-click a signal in the **Legend** tab view a pop-up menu opens with options that are described in [Legend](#), p.74.

Properties Tab View

You can view and change the physical appearance properties for the **Plot** window you are in by clicking the **Properties** tab in the **Signals Bar** (but not the signals themselves - for that see [7.3 Signal Properties Dialog Box](#), p.123).

Windows Host



UNIX Host

Signals Bar	
Property	Value
Plot Properties	
Y Scale	3
Y Offset	1.5
Display Accuracy	4
Min Grid Spacing	0.25
Resolution	1
Dump Properties	
Resolution (secs)	1
History Limit	500
Display Accuracy	4
Monitor Properties	
Resolution (secs)	1
WriteBack	True
Display Accuracy	4
WriteBack Warnings	True
Display Int as Hex	False
Display Char as Hex	True
Active Signals	
Properties	

Physical appearance properties options in both Windows and UNIX hosts are:

- **Y Scale**
Controls the height of the plot grid in units. For example, when first started, Y Range defaults to 3 and the plot displays the Y-axis from 1.5 to -1.5.
- **Y Offset**
Controls the unit value for the top coordinate on the plot. When first started, it defaults to 1.5, and the default Y value of 1.5 is the top value. This value changes automatically if you use the **Zoom to Fit** command.
- **Display Accuracy**
Controls the number of significant digits displayed in the grid line markers. Set this property to the number of places you want displayed to the right of the decimal point. Enter a value between 0 (for integer numbers) and 6. The default is 4.

- **Resolution**

Permits plotting of only a subset of the collected data points. This is useful for increasing rendering speed. A divisor value of n causes only every n th point to be plotted. Thus, if **1000** points are being collected and the **Resolution** is **5**, then **200** points are displayed. Of course, you may not want to reduce resolution if your data can contain glitches or high-frequency phenomena. The default is **1**.

- **Grid Spacing**

Specifies the minimum distance (in pixels) allowed between the grid lines. Increasing this number decreases the number of grid lines, decreasing this number increases the number of grid lines. Enter a value between **5** and **500**. The default is **100**.

- **Dump resolution**

Controls how often to refresh the values in the **Mini Dump** window, in seconds. The default is **1.000000**.

- **Dump history limit**

Controls how many lines of historical data to maintain in the **Mini Dump** window (**0** = display all). The default is **1000**.

- **Monitor resolution**

Controls how often to refresh the values in the **Mini Monitor** window. The default is **1.000000**.

- **Allow writeback**

Select **True** to create a **writeback** column for writing modified signal values back out to the target. The default is **False**.

The procedure for using writeback in the Mini-Monitor window is exactly the same as for the Monitor window. For detailed instructions, see [10.3 Writing Data to the Target](#), p.181.

Additional options for **Windows** hosts only are:

- **X Scale**

Controls the width of the plot grid in units. For example, when first started, X Range defaults to **3** and the plot displays the X-axis from **1.5** to **-1.5**.

- **X Offset**

Controls the unit value for the left-most coordinate on the plot. When first started, it defaults to **1.5**, and the default X value of **1.5** is on the right side. This value changes automatically if you use the **Zoom to Fit** command.

- **Snap measures to signals**

Controls whether or not measures are snapped to the signal lines on the grid. Measures are described in [On-grid Measurements](#), p.78. The default is **True**.

- **Maximum snap distance**

Controls how far away (in pixels) you can start a measure and still have it snap to the plot line. Measures are described in [On-grid Measurements](#), p.78. The default is **20**.

- **Background image**

Allows you to specify special images to be displayed in the graphing area, with the graph itself superimposed on top. The default is **None**.

Additional options for UNIX hosts only are:

- **Display Accuracy (Dump and Monitor)**

Controls the number of significant digits displayed in the grid line markers. Set this property to the number of places you want displayed to the right of the decimal point. Enter a value between **0** and **6**. The default is **4**.

- **Resolution**

Permits plotting of only a subset of the collected data points. This is useful for increasing rendering speed. A divisor value of *n* causes only every *n*th point to be plotted. Thus, if **1000** points are being collected and the **Resolution** is **5**, then **200** points are displayed. Of course, you may not want to reduce resolution if your data can contain glitches or high-frequency phenomena. The default is **1**.

- **WriteBack**

Select **True** to create a **writeback** column where you can write modified signal values back out to the target. The default is **True**.

The procedure for using writeback in the Mini-Monitor window is exactly the same as for the Monitor window. For detailed instructions, see [10.3 Writing Data to the Target](#), p.181.

- **WriteBack Warnings**

Controls whether or not a warning is displayed before each writeback attempt. The default is **True**.

- **Display Int as Hex**

Controls whether or not to display integer values in hexadecimal. The default is **True**.

- **Display Char as Hex**

Controls whether or not to display char variables in hexadecimal. The default is **False**.

Any changes you make in this window have no effect on any other open **Plot** windows.

To change the defaults used when new **Plot** windows are created, use the **File > Preferences** menu command (or the **Preferences** button), described in [3.3.12 Preferences](#), p.53.

Legend Window (UNIX Hosts Only)

In a UNIX host, the **Legend** window shows you what color is being used for each selected signal. You can also use the Legend window to select and clear signals. The Legend window performs the same function in a UNIX host GUI that the Legend tab view in the Signals Bar performs in a Windows host GUI.

For detailed information, see [Legend Window \(UNIX Hosts Only\)](#), p.27.

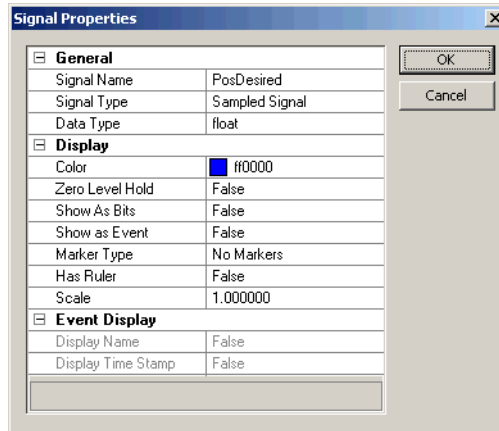
Strip Chart

This feature, turned on or off using the **Plot > Strip Chart** menu command (or the **Strip Chart** toolbar button; see [Plot Window Toolbar](#), p.41), changes the behavior of the graph area to present a continuously scrolling plot (instead of repainting the plot every 10 seconds). The overall appearance of the **Plot** window does not change in any way. Any snapshots you may be displaying on the grid when you select **Strip Chart** become unselected; snapshots remain unavailable while **Strip Chart** is selected.

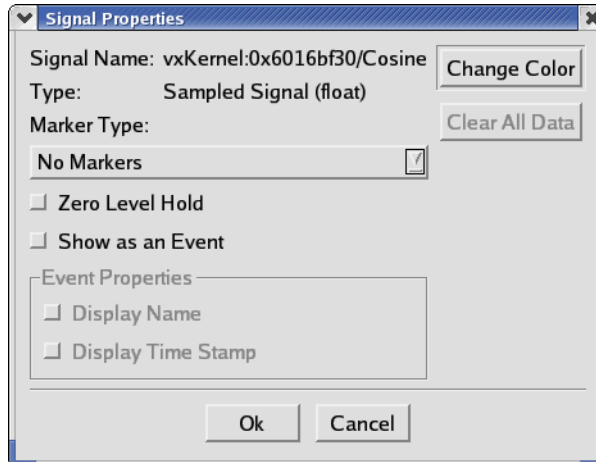
7.3 Signal Properties Dialog Box

The physical characteristics of the line currently being used to plot each signal in this window can be configured using the **Signal Properties** dialog box.

Windows Host



UNIX Host



The dialog box can be opened in any of the following ways:

- Right-click a signal in the **Signals** tab view.

- Right-click a signal in the **Legend** tab view.
- Right-click a **signal trace** in the data-display area.

In the pop-up menu select **Properties**. The **Signal Properties** dialog box opens.

Signal line properties options in both Windows and UNIX hosts are:

- **Signal Name and Type**

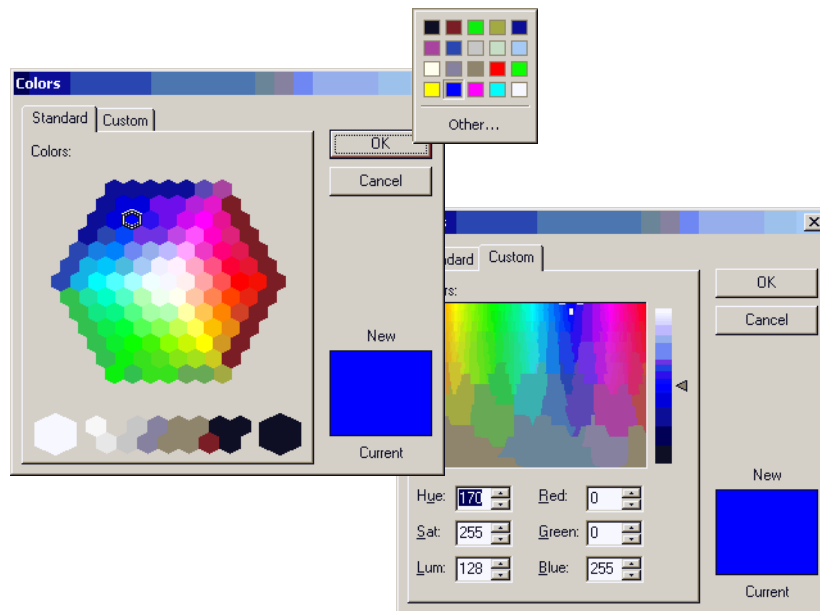
The first two lines are the name of the signal and its type (as shown in the **Signal Tree**).

- **Data Type**

This is the C++ data type of this signal (program variable). For a list of allowable types, see [Table 16-1](#).

- **Color**

Clicking on the current value opens a small color palette with a basic color selection from which to choose.



If you would like a color that is not on this small color palette, you can click **Other** to open the **Colors** dialog box where you can select a new color from the

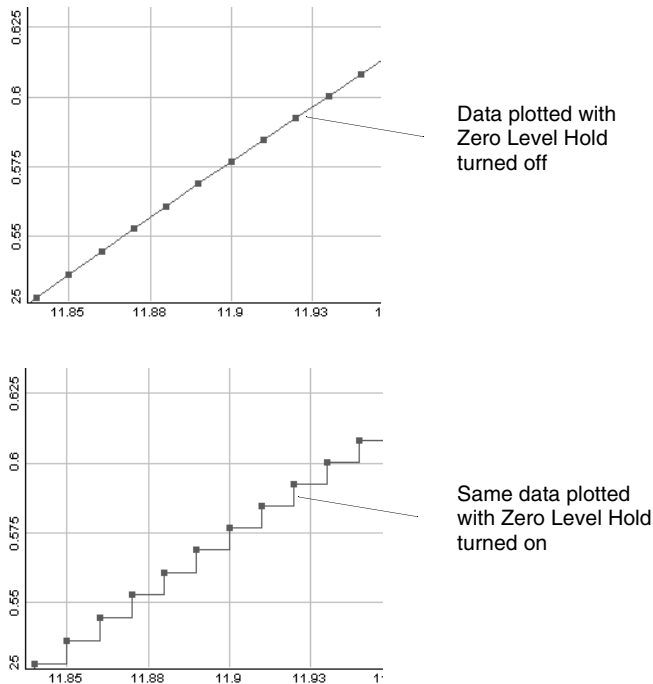
larger selection in the **Standard** tab view, or you can create an entirely new color using the **Custom** tab view.



NOTE: An alternative method for changing colors of plotted lines on the **Plot** window graph is given in the discussion of colors preferences in [Colors View](#), p.56.

- **Zero Level Hold**

Select **True** from the drop-down menu to hold the value of the signal until the next sample arrives. To demonstrate this feature, consider the sampling of a sine wave at a rate of 100Hz.



In the upper view in this figure, with **Zero Level Hold** turned off (= **False**), the data points are connected as usual with straight lines, even though there is no information about the actual values between data points. In the lower view, the same data, but with the **Zero Level Hold** feature turned on (= **True**), shows how the line plotted between the same data points is now a straight horizontal

line of the same value as the previous data point, until the next data point comes in, at which time the plot line goes vertical up to that value.

- **Show as Event**

Select **True** from the drop-down menu to cause samples to be marked with vertical lines instead of connecting the individual samples with lines. For more information, see [7.5 Displaying Events](#), p. 130.

- **Marker Type**

This drop-down list allows you to select a different symbol (or marker) to be plotted for each sample of this signal. The choices are:

- Square — " □ "
- Plus — " + "
- Diamond — " ◇ "

- **Display Name**

If **True**, then the name of the event is displayed in the plot. The default is **False**.

- **Display Time Stamp**

If **True**, then the event time stamp is displayed in the plot. The default is **False**.

Additional options for **Windows** hosts only are:

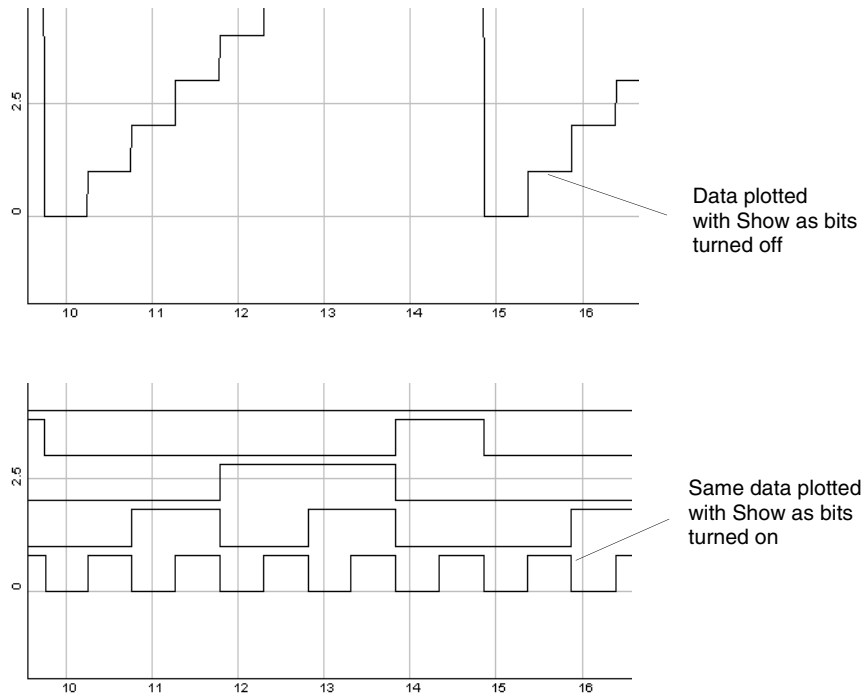
- **Data Type**

This is the C++ data type of this signal (program variable). For a list of allowable types, see [Table 16-1](#).

- **Show as Bits**

Select **True** from the drop-down menu to cause a signal (program variable) being plotted to be represented as a sequence of bits, each bit with its off and on (0 and 1) values indicated. The bits are rendered individually by discrete horizontal lines distributed up the Y axis from bit position 0 starting at the bottom.

To show this feature, a **Derived Signal**, created using a single 8-bit integer, is incremented through values from 0 to 15 and back to 0 again at the rate of two increments per second, and then repeated continuously until stopped. The modified plot is shown here.



The top graph shows the value of the variable with **Show as bits turned off** (= **False**), and the lower graph shows the same data with **Show as bits turned on** (= **True**). You can easily observe the pattern of the individual bits (horizontal lines) making up the 16-bit integer, showing their 0 and 1 positions with respect to time.

This feature works equally well with all signal types, but the resulting bit patterns may be more difficult to decipher depending on the complexity of the signal.

- **Has Ruler**

This True/False option displays a separate ruler with colored numbers matching the color of the signal on the left (Y) axis. If you have, say, three signals with rulers on, there are three separate color-coded rulers on the left plot boundary. The default is **False** (= no rulers on).

- **Scale**

Each plotted value is scaled (multiplied) by the value you enter. The default is **1.000000**.

Additional options for **UNIX** hosts only are:

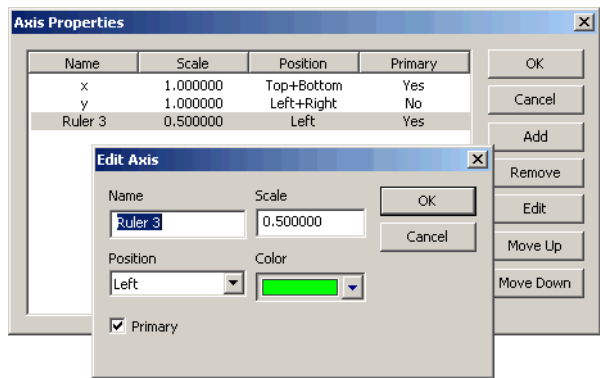
- **Clear All Data (button)**

Resets all the signal properties to Data Monitor default values.

When you have finished changing parameters, select **OK** to save your changes and exit the dialog box. Select **Cancel** to exit the dialog box without saving any changes you have made.

7.4 Axis Properties Dialog Box (Windows Hosts Only)

In a Windows host, the **Axis Properties** dialog box is opened with the **View > Axis Properties** menu item (or the **Axis Properties** toolbar icon).



Using this dialog box you can modify the appearance of the **Plot** window graphing area, including X and Y axis text spacing, text colors, and certain other properties. You can also add new axis markings on any edge, remove existing axis markings from any edge, or reposition existing axis markings.

In addition to the usual **OK** and **Cancel**, this dialog box facilitates the following actions by means of the following buttons:

- **Add**

This button opens the Edit Axis dialog box, where you can specify parameters for a new axis label.

- **Remove**

Select any axis label in the **Axis Properties** dialog box and delete it from the graph area using this button. Only one axis label at a time can be selected for removal.

- **Edit**

With an existing axis label selected in the **Axis Properties** dialog box, selecting this button opens the Edit Axis dialog box shown above. In this dialog box you can modify the following characteristics of the axis label:

- **Name**

Accept the default name, or enter your choice of names by typing over.

- **Scale**

Scales the marker numbers up or down with respect to the actual plot boundary values. The number printed is the actual value divided by the **Scale** value.

- **Position**

Determines where the axis label is to be placed, with drop-down menu values that are combinations of right and left, or top and bottom.

- **Color**

Selects the color for the text in your axis marker. Colors are selected as described above in [7.3 Signal Properties Dialog Box](#), p.123

- **Primary**

This checkbox, when selected, causes gridlines to be drawn only for this signal, as well as any other signals for which this check box has been selected.

- **Move Up**

Selecting a row in the dialog box and clicking this button moves this axis marker in (closer) toward the graphing area than all the other rows below it in the same (**left/right** or **top/bottom**) axis.

- **Move Down**

Selecting a row in the dialog box and clicking this button moves this axis marker out (farther away) from the graphing area than all the other rows above it in the same (**left/right** or **top/bottom**) axis.

The axis markers you set up with this dialog box will persist, even across Data Monitor sessions, until you change them again.

7.5 Displaying Events

This section describes how events, collected using the Data Monitor API, are displayed. Event collection routines include **ScopeEventsCollect()** (collects an event identifier and the value of a variable) and **ScopeEventsMessage()** (collects a string).

For a target, events appear under the **Event** tree branch in the **Signals Bar**. This tree contains a list of all the events that were thrown. By default, events that collect a value (from **ScopeEventsMessage()**) are displayed much like normal signals. The values collected when a certain event is thrown are treated like a sample and the samples are joined by lines to make signals. Alternatively, the **Zero Order Hold** option can be used to join samples in a step-wise manner (the last value is held until a new sample appears).



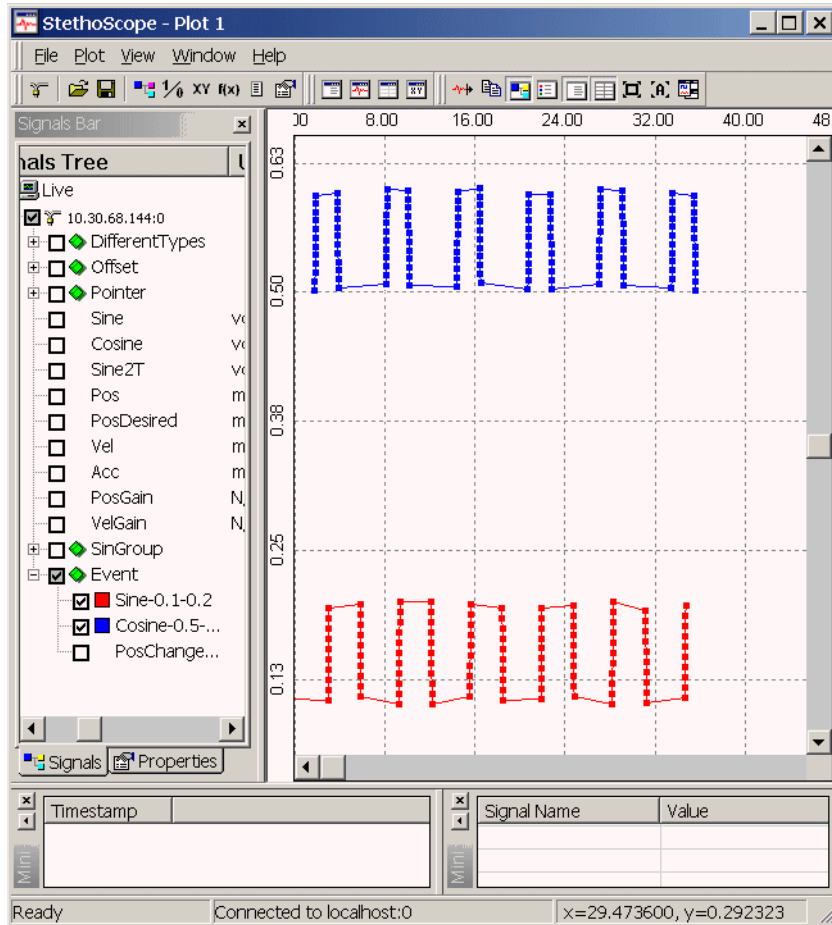
NOTE: It is a known problem that if you select **events** and **signals** to be plotted simultaneously, the resulting traces displayed over a period of time begin to diverge, and the plot becomes compromised and lacking integrity. Do not try to plot these two data types on the same graph.

Events are displayed according to their mode of collection, in three different categories:

- **Events Collected as Signals**
- **Events Collected as Markers**
- **Events Collected as Messages**

Events Collected as Signals

This example shows the display of events collected as **signals**.

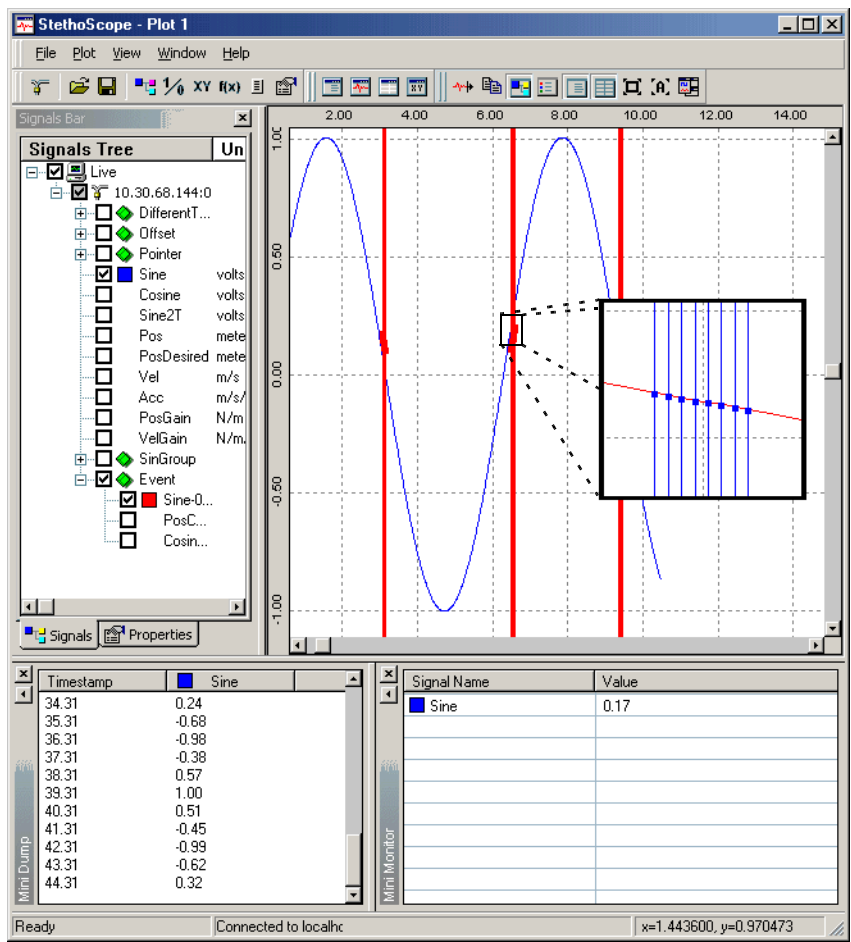


Events Collected as Markers

Events that collect a value (from `ScopeEventsCollect()`) can also be displayed as vertical lines, or **markers**. A marker appears at the temporal location on the plot corresponding to the timestamp that was collected with the event. When events

are displayed as vertical lines, the name of the event (**event ID**), and the timestamp can be displayed by choosing options **Display Name** and **Display Timestamp** respectively.

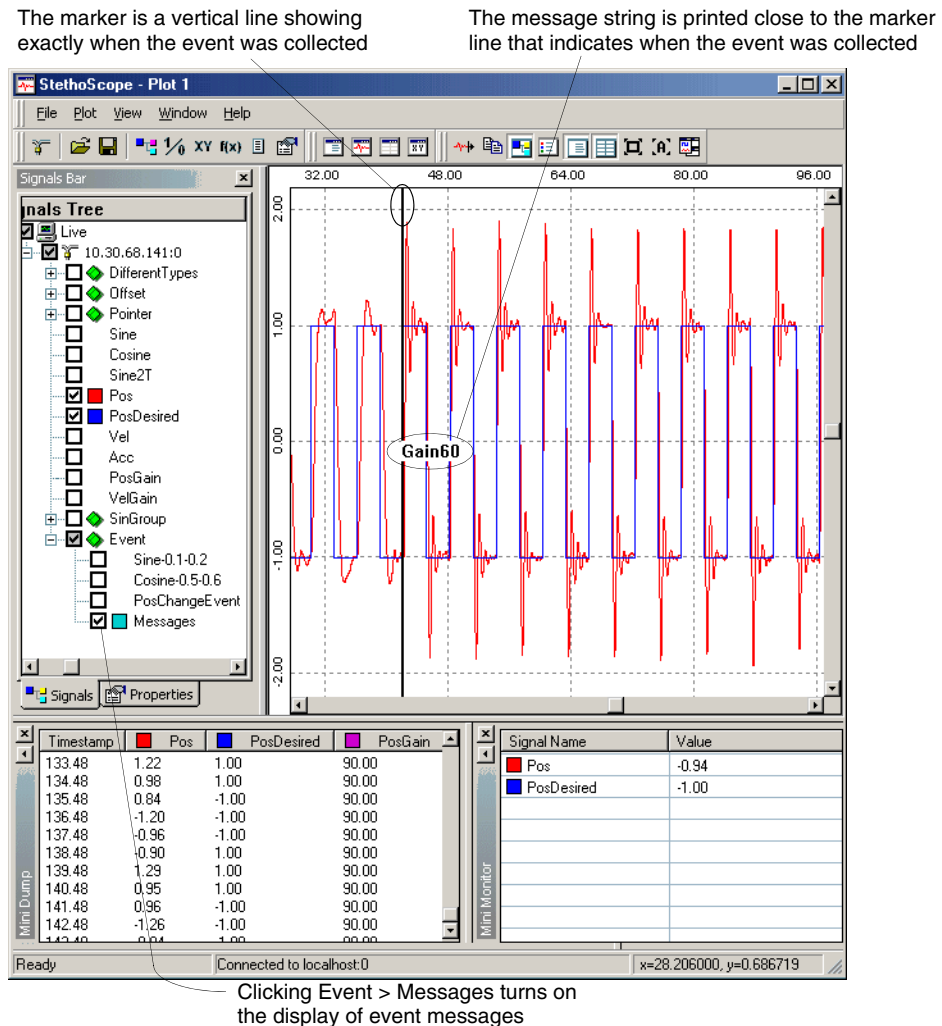
This example shows the display of events collected as a value but displayed as **markers**.



Events Collected as Messages

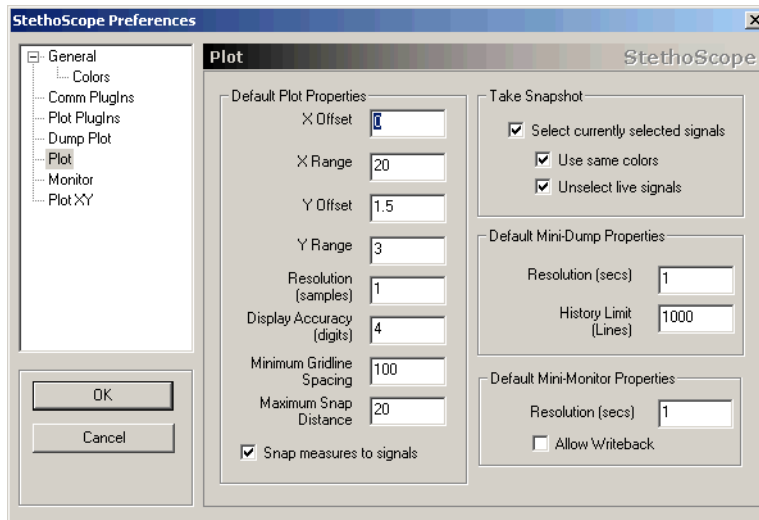
Event messages, collected using **ScopeEventsMessage()**, displayed as character strings under the **Event** tree. Turning on event messages causes vertical lines to be displayed, representing the temporal location of the occurrence of the event. The message itself is also displayed.

This example shows the display of events collected as **messages**.



7.6 Setting New Plot Window Preferences

The **Data Monitor Preferences** dialog box (also see [3.3.12 Preferences](#), p.53) is opened with the **File > Preferences** menu command (or the **Preferences** toolbar button - see [Main Toolbar](#), p.40).



It allows you to set default parameters used when any new data-display window is opened (whereas the **properties** described in [7.3 Signal Properties Dialog Box](#), p.123 above, apply only to the currently open window in which the properties are set).

This dialog box also allows you to control some aspects of what happens when a snapshot is taken. One additional preference in particular that affects the **Plot** window is **Colors** (see [Colors View](#), p.56).

To modify these preferences, open the **Data Monitor Preferences** dialog box, using the **File > Preferences** menu command (or the **Preferences** button), then click **Plot** in the left panel to display and set the following parameters.



NOTE: The Windows and UNIX host versions of this dialog box have exactly the same parameters except as noted below.

Default Plot Properties Panel

- **X Offset (Windows hosts only)**

Controls the unit value for the left-most coordinate on X axis of the plot. When first started, it is 0. This value changes automatically if you use the **Zoom to Fit** command.

- **X Range (Windows hosts only)**

Controls the width of the plot grid in units. For example, when first started, X Range defaults to 20, and the plot displays the X-axis from 0 to 20.

- **Y Offset**

Controls the unit value for the top coordinate on the plot. When first started, it is 1.5, and the Y value of 1.5 is the top value. This value will change automatically if you use the **Zoom to Fit** command.

- **Y Range**

Controls the height of the plot grid in units. For example, when first started, Y Range defaults to 3 and the plot displays the Y-axis from 1.5 to -1.5.

- **Resolution (samples)**

Permits plotting of only a subset of the collected data points. This is useful for increasing rendering speed. A divisor value of *n* causes only every *n*th point to be plotted. Thus, if 1000 points are being collected and the **Resolution** is 5, then 200 points are displayed. Of course, you may not want to reduce resolution if your data can contain glitches or high-frequency phenomena. The default is 1.

- **Display Accuracy (digits)**

Controls the accuracy of the grid line markers. Set this property to the number of places to the right of the decimal point to use in the grid markers. Enter a value between 0 (for whole numbers) and 6. The default is 4.

- **Minimum Grid Line Spacing**

Specifies the minimum distance (in pixels) allowed between the grid lines. Increasing this number decreases the number of grid lines, decreasing this number increases the number of grid lines. Enter a value between 5 and 500. The default is 100.

- **Maximum Snap Distance**

Controls how far away (in pixels) you can start a measure and still have it snap to the plot line. Measures are described in [On-grid Measurements](#), p.78. The default is 20 pixels.

- **Snap Measure to Signals**

Check box controls whether or not measures are snapped to the signal lines on the grid. Measures are described in [On-grid Measurements](#), p.78.

Take Snapshot Panel

Unlike all other preferences, which only impact data-display windows you may create later, these preferences take effect immediately. For more information, see [11. Working with Snapshots](#).

- **Select currently selected signals**

If selected, the same signals currently selected in the live buffer are also selected in the snapshot.

- **Use same colors**

If selected, the same colors are assigned to the signals in the snapshot. If you want the snapshot to use different colors for each signal than are used for live data, clear this check box.

- **Unselect live signals**

If selected, the live signals will become unselected when the snapshot is taken.

Default Mini-Dump Properties Panel

These properties apply to the **Mini-Dump** window inside a new **Plot** window. They have no effect on any full-size **Dump Plot** windows.

- **Resolutions (secs)**

Controls how often (in seconds) to refresh the values in the **Mini-Dump** window. The default is 1.

- **History Limit (Lines)**

Controls how many lines of historical data to maintain in the **Mini-Dump** window. The default is 1000.

Default Mini-Monitor Properties Panel

These properties apply to the **Mini-Monitor** window inside a new **Plot** window. They have no effect on any full-size **Monitor** windows.

- **Resolution (secs)**

Controls how often (in seconds) to refresh the values in the **Mini-Monitor** window. The default is **1**.

- **Allow Writeback**

Controls whether or not you can use the **Mini-Monitor** window to write modified signal values back out to the target. The default is **False**.

- **Enable Warnings (UNIX hosts only)**

Controls whether or not warning messages are sent to the **Console** window when writeback values you entered are about to be processed, allowing you a chance to cancel the operation. The default is **selected**.



NOTE: Note that no currently open data-display windows will be modified by configuring any of these parameters - only ones you open after configuring.

8

The Plot XY Window

- 8.1 Introduction 139
- 8.2 Creating XY Signal Pairs 140
- 8.3 Plot XY Window Tour 141
- 8.4 Signal Properties Dialog Box 154
- 8.5 Setting New Plot XY Window Preferences 160

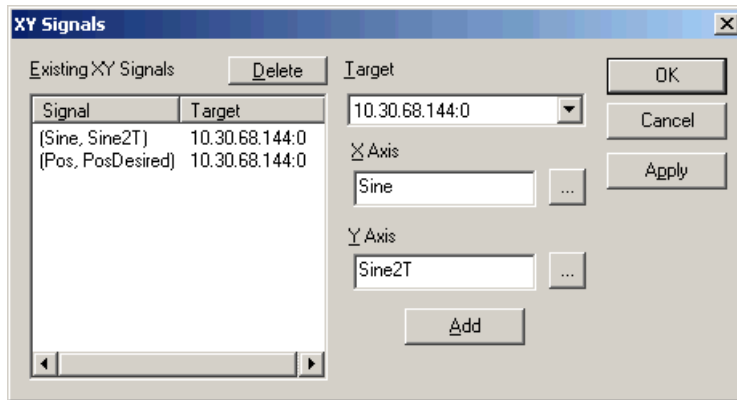
8.1 Introduction

The **Plot XY** window is used to graph pairs of signals against each other. In most respects, it is very similar to the **Plot** window. The primary difference is that only XY signal pairs show up in the **Signals Tree** to be plotted. XY signal pairs must be created in order to have anything to plot. This chapter describes the process of creating XY signal pairs, as well as the **Plot XY** window itself.

8.2 Creating XY Signal Pairs

You must create XY signal pairs before you can use the **Plot XY** window.

Signal selection for **Plot XY** windows differs from signal selection for the other types of data-display windows, in that each plotted line requires two signals (an XY signal pair). These XY signal pairs are created using the **XY Signals** dialog box.



Creating a Signal Pair

Open the dialog box using the **File > XY Signals** menu command (or the **XY Signals** toolbar button - see [Main Toolbar](#), p.40). To create a signal pair in this dialog box, follow these steps:

1. Choose a target from the **Target** drop-down list box.
2. Choose the signal to plot on the X Axis by using the **X Axis** drop-down list. The list contains only active signals for the selected target. Double-click the desired signal.
3. Choose another signal to plot on the Y Axis by using the **Y Axis** drop-down list. This list also contains only active signals for the selected target. Double-click the desired signal.
4. Click **Add**.
5. For each **XY Signal** pair you want to create and be able to see in the **Plot XY** window, repeat steps 1 through 4 above.

6. When you are done, click **OK**, **Cancel**, or **Apply**.
Apply saves your selections, but leaves the dialog box open for more.
OK saves your selections and closes the dialog box.

Deleting a Signal Pair

To delete a signal pair, follow these steps:

1. Select the signal pair from the **Existing XY Signals** list.
2. Click **Delete**.

8

Modifying a Signal Pair

To modify a signal pair, the only option is to delete the pair, then add a new one.



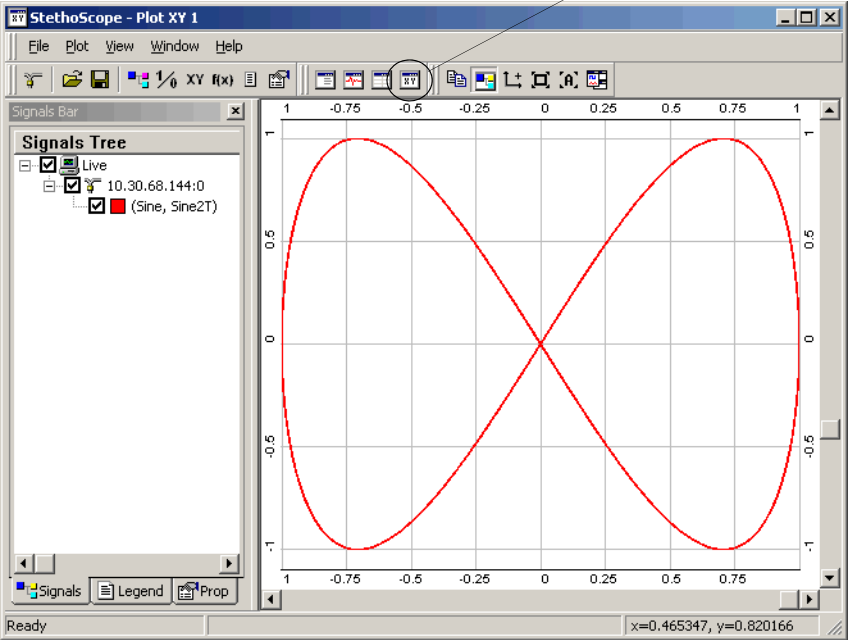
NOTE: The XY Signals dialog box in a UNIX host has exactly the same elements, but some are in slightly different locations in the dialog box. Therefore, the Window host version description above also adequately describes the UNIX version.

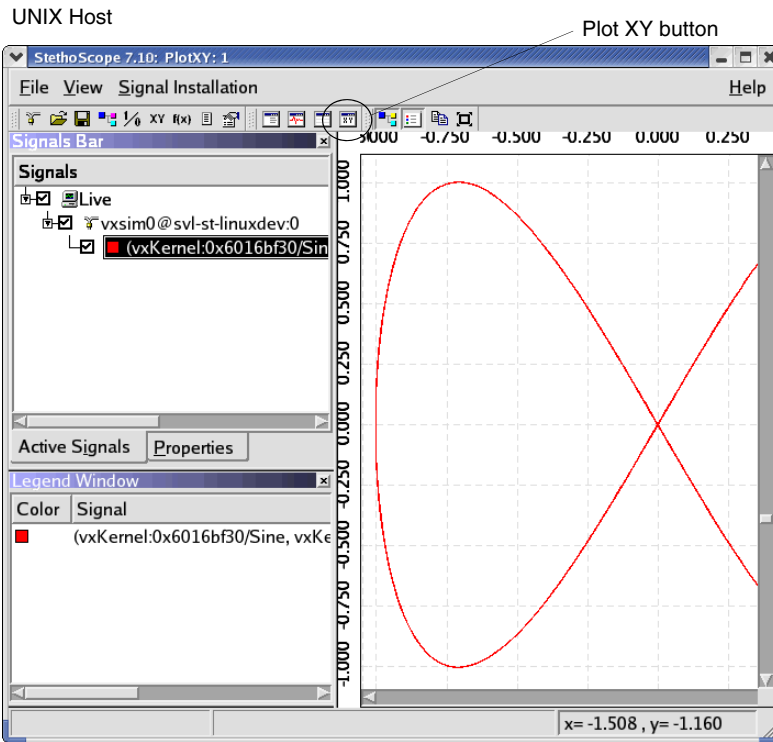
8.3 Plot XY Window Tour

Open the **Plot XY** window using the **File > Plots > Plot XY** menu command (or the **Plot XY** toolbar button - see [Plots Toolbar](#), p.41).

Windows Host

Plot XY button





Displaying Signal Parameters

To display signal parameters, follow these steps:

1. If you do not already have a **Signals Bar** displayed in your **Plot XY** window, open one using the **File > Signals Bar** menu command (or the **Signals Bar** toolbar button - see [Plot Window Toolbar](#), p.41).
2. Make sure the **Signals** tab is selected in the **Signals Bar**, so that a **Signals Tree** is displayed.
3. Use the **Signals Tree** to select which signals you want to display in the graph. For details on using **Signals Trees**, see [4. Using the Signal Manager](#). If you do not see any signals in the **Signals Tree**, use the **File > XY Signals** menu command (or the **XY Signals** toolbar button - see [Main Toolbar](#), p.40) to open the **XY Signals** dialog box and create signal pairs, as described in [8.2 Creating](#)

[XY Signal Pairs](#), p.140. Selecting a signal from the **Plot XY** window **Signals Tree** causes the following actions:

- The signal data is plotted in the window, using the next available plot color. You can select a different color using the **Signal Properties** dialog box (see [8.4 Signal Properties Dialog Box](#), p.154), or the **Preferences** dialog box (see [3.3.12 Preferences](#), p.53).
- The **Legend** tab view (or the **Legend** window in a UNIX host) is updated to include the selected signal. The Legend shows you what color is being used for each selected signal. You can also use the Legend to select and clear signals.

The Legend tab view, and the Legend window, are the same as in the Plot window. The Legend tab view is described in [Legend Tab View \(Windows Hosts Only\)](#), p.117, and the Legend window is described in [Legend Window \(UNIX Hosts Only\)](#), p.122.

Most of the functionality in the **Plot XY** window is identical to that in the **Plot** window, except there is no **Strip Chart** option, **Mini Dump** window, or **Mini Monitor** window. For more information on common **Plot XY** window features, refer to the following sections:

- [3.2 Toolbars](#), p.40
- [Plot Menu Item \(Windows Hosts Only\)](#), p.69
- [View Menu Item](#), p.70
- [3.5 Pop-up Menus](#), p.71

Popup Menu

When you right-click anywhere in the plot area, the **On-grid** pop-up menu opens with several options pertaining specifically to the plot area. These additional options are described in detail in [3.5 Pop-up Menus](#), p.71.

Screen Operations

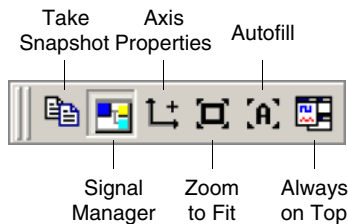
The screen operations outlined for the **Plot** window also apply to the **Plot XY** window. These operations are described in [3. Data Monitor Features](#), and include:

- [Zooming](#), p.76
- [Markers](#), p.76
- [Annotations](#), p.77
- [Panning](#), p.77
- [On-grid Measurements](#), p.78

Toolbar

In a default **Plot XY** window, the first two toolbars are shown with the menu items as described in Sections [Main Toolbar](#), p.40 and [Plots Toolbar](#), p.41. But the right-most toolbar is specific to the **Plot XY** window. All the toolbars (as well as the menu bar) are dockable, which means you can move them to other locations, on or off the window, simply by dragging them. Each of the toolbars can be displayed or hidden independently using the **View** menu item (see [View Menu Item](#), p.146).

This **Plot XY** window toolbar is a subset of the one shown and described in detail in [Plot Window Toolbar](#), p.41.

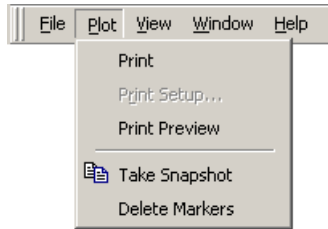


Menu Bar

Some of the menu bar items are discussed in detail in [3.3 File Menu Item](#), p.43, where it is mentioned that the **File**, **Window**, and **Help** menu items are consistently the same across all data-display windows. For the **Plot XY** window, however, the **Plot** and **View** menu items contain some commands that are unique to the **Plot XY** window. These menu items are described in detail in the following sections.

Plot Menu Item (Windows Hosts Only)

In a Windows host, the **Plot** menu item contains commands for working with the data displayed in the **Plot XY** window.



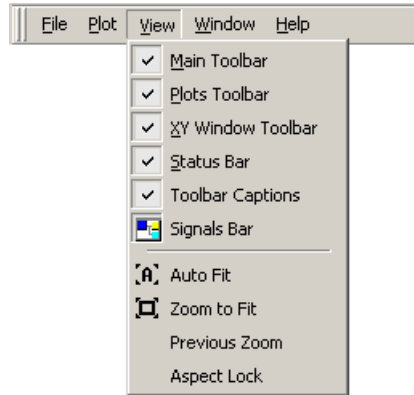
The **Plot** menu options for the **Plot XY** window are:

- **Print**
Prints the signal traces in the data-display window. It opens a **Print** dialog box where you can configure your printer options.
- **Print Setup**
Allows you to select printer parameters and characteristics before printing.
- **Print Preview**
Lets you see, on a print mock-up screen, what the printed page will look like before actually printing it.
- **Take Snapshot**
Saves a copy of all the active signals (not just the selected signals) for all connected targets. For more information, see [11.2 Utilizing Snapshots](#), p.185.
- **Delete Markers**
Deletes all markers, annotations, and measures from the grid area. For more information, see [Markers](#), p.76.

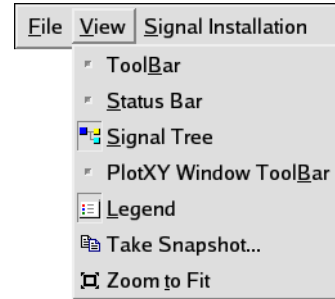
View Menu Item

The **View** menu item, in both a Windows and a UNIX host, contains commands for displaying toolbars and auxiliary views in the **Plot XY** window, as well as some **Plot XY** window-specific commands.

Windows Host



UNIX Host



The following menu options are available in both Windows and UNIX hosts:

- **PlotXY Window Toolbar**

Controls whether the toolbar representing a selection of items from the **File** menu is displayed on Data Monitor's toolbar. For more information, see [3.2 Toolbars](#), p.40.

- **Status Bar**

Controls whether the status line along the bottom of the window is visible. For more information, see [3.7 Status Bar](#), p.78.

- **Zoom to Fit**

Changes the scales and offsets so that all the signals fit and take up the entire plot window. The scales and offsets remain at these values until **Zoom to Fit** is selected again. This command is only available in the **Plot XY** windows. See also **Previous Zoom**.

Additional options for **Windows** hosts only are:

- **Main Toolbar**

Controls whether the toolbar representing a selection of items from the **File** menu is displayed on Data Monitor's toolbar. For more information, see [3.2 Toolbars](#), p.40.

- **Plots Toolbar**

Controls whether the toolbar representing a selection of items from the **File > Plot** menu command is displayed on Data Monitor's toolbar. For more information, see [3.2 Toolbars](#), p.40.

- **Toolbar Captions**

Controls whether the names of the panels (**Signals Bar**, **Mini-Dump**, and **Mini-Monitor**) are displayed in the **Plot** window above their respective invocations.

- **Signals Bar**

Controls whether a **Signals Bar** panel appears in the window. The **Signals Bar** includes tabs for **Signals** and **Properties**.

- **Auto Fit**

Causes the plotting area to be scaled dynamically and automatically to the extremes of the data in the Y direction being displayed in the window. It thus allows all data points to appear on the plot. It toggles on or off, but when on, it essentially has the effect of using the **Zoom to Fit** button continuously. This command is only available in the **Plot XY** window.

- **Previous Zoom**

Reverses the action of the last zoom action in a **Plot** or **Plot XY** window. Note that when you use the **Zoom to Fit** command, the history of all previous zoom actions is lost, therefore this command has no effect immediately after a **Zoom to Fit** command. Also, if no previous zoom has been set, this item is greyed out and unavailable.

- **Aspect Lock**

When selected, it causes the plot window aspect ratio to be maintained when you zoom in or out on the plot.

Additional options for **UNIX** hosts only are:

- **Toolbar**

Controls whether the toolbar representing a selection of items from the **File** menu is displayed on Data Monitor's toolbar. For more information, see [3.2 Toolbars](#), p.40.

- **Signal Tree**

Controls whether the Signal Manager window is displayed. For more information, see [4. Using the Signal Manager](#).

- **Legend**

Controls whether the Legend window is displayed. For more information, see [Legend Window \(UNIX Hosts Only\)](#), p.27.

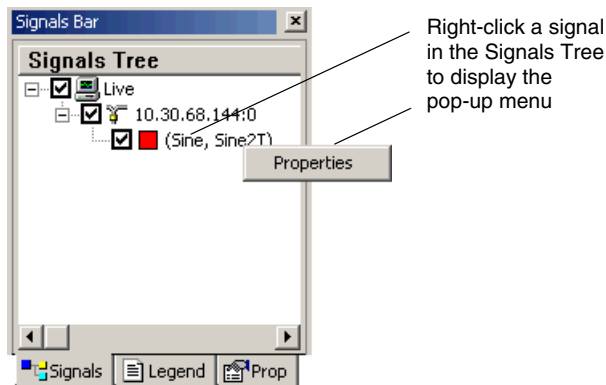
- **Take Snapshot**

Saves a copy of all the active signals. You can display the snapshot in the **Plot** window along with real-time data. For more information, see [11.2 Utilizing Snapshots](#), p.185.

Signals Bar

8

The **Signals Bar** is a sub-window in the Data Monitor GUI that allows you to view and configure information that controls what is plotted in the graph area.



To open a **Signals Bar** if one is not already displayed, use the **View > Signals Bar** menu command (or the **Signals Bar** toolbar button - see [Plot Window Toolbar](#), p.41). Note that only **XY signal pairs** are displayed in the **Signals Bar** for this window.

The **Signals Bar** contains the following tab views:

- [Signals Tab View](#)
- [Legend Tab View \(Windows Hosts Only\)](#)
- [Properties Tab View](#)

Signals Tab View

The **Signals Tree** in the **Signals** tab view (see the figure above) displays all the signals available to be plotted. They are shown in an expandable tree structure containing signals that have been installed by the **Signal Manager** (see [4. Using the Signal Manager](#)). Traces on the graph are color-coded to the signals selected in the **Signals Tree**.

The **Signals Tree** is displayed by default when you open a **Plot XY** window, but it may be re-displayed at any time by clicking the **Signals** tab at the bottom of the sub-window.

The **Signals Tree** allows you to easily locate any signal that has been installed and add it to the graph of signal traces. Each node of the tree (and hence each signal) has a check box. Selecting the check box for an individual signal adds that signal to the graph, or, conversely, clearing a check box removes that signal from the graph. If you select a node check box, all the signals belonging to that node (including their nodes below it) are selected at once and their signals are added to the graph. Conversely, clearing a node check box removes all signals below that node from the graph with a single click.

If a node check box is selected, but has a grey fill, it indicates that some, but not all, of the signals belonging to that node are selected and displayed on the graph. You may have to scroll down to see which ones are selected.

The **Signals Tree** is expanded or collapsed using the icon to the left of each node check box as follows:

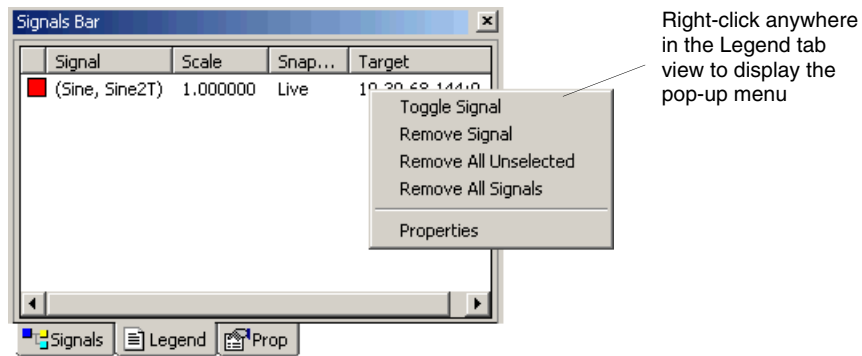
- “+” = **collapsed**; select to expand down to the next node(s).
- “-” = **expanded**; select to collapse up to this node.

The **Units** column in the **Signals** tab view shows the physical measurement units of each component of the **XY Signal** pair.

When you right-click a signal in the **Signals** tab view, a pop-up menu opens, currently containing a single menu item, as shown in the figure above. The **Properties** menu item opens the **Signal Properties** dialog box where you can configure parameters that affect the appearance and behavior of the specific signal your cursor is on. The **Signal Properties** dialog box is described in detail in [8.4 Signal Properties Dialog Box](#), p.154.

Legend Tab View (Windows Hosts Only)

In a Windows host, the **Legend** tab view shows you what color is assigned to each signal on the plot. It can also be used to select which signals to plot. This figure is an example of the **Legend** tab view in a **Plot XY** window.



NOTE: In a UNIX host, the **Legend** appears in a separate window, rather than in a tab view. For details, see [Legend Window \(UNIX Hosts Only\)](#), p.27.

The columns in the **Legend** tab view display the current option settings for each **Signal**, including:

- **Scale**
The scale factor applied to the values for each component signal to allow them to be easily viewed on the same graph with other **XY Signal** pairs.
- **Snapshot**
Whether the data is live, or from a snapshot.
- **Target**
The name and scope index of the target to which you are connected.

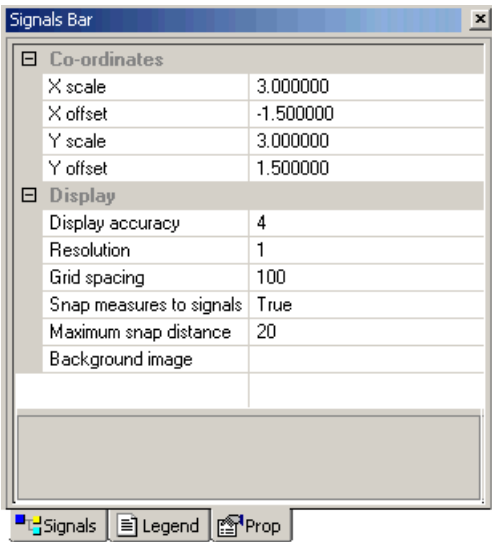
When you right-click a signal in the **Legend** tab view a pop-up menu opens with options that are described in [Legend](#), p.74.

You can change the colors used for plot lines using the **File > Preferences** menu command (see [Colors View](#), p.56), or by using the **Signal Properties** dialog box (see [8.4 Signal Properties Dialog Box](#), p.154).

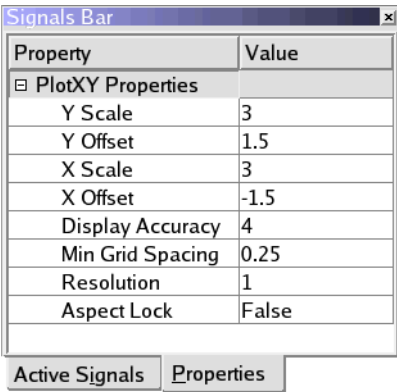
Properties Tab View

You can view and change the physical appearance properties for the **Plot XY** window you are in by clicking the **Properties** tab in the **Signals Bar** (but not the signals themselves; for that, see [8.4 Signal Properties Dialog Box](#), p.154).

Windows Host



UNIX Host



This tab view accesses properties for the graphic display window environment (for the signals themselves, use the **Signal Properties** dialog box, described in [8.4 Signal Properties Dialog Box](#), p.154).

Physical appearance properties options in both Windows and UNIX hosts are:

- **X scale**

Controls the width of the plot grid in units. For example, when first started, X scale defaults to **3** and the plot displays the X-axis from **1.5** to **-1.5**.

- **X offset**

Controls the unit value for the left-most coordinate on the plot. When first started, it is **1.5**, and the X value of **1.5** is on the right side. This value changes automatically if you use the **Zoom to Fit** command.

- **Y scale**

Controls the height of the plot grid in units. For example, when first started, Y Scale defaults to **3** and the plot displays the Y-axis from **1.5** to **-1.5**.

- **Y offset**

Controls the unit value for the top coordinate on the plot. When first started, it is **1.5**, and the Y value of **1.5** is the top value. This value changes automatically if you use the **Zoom to Fit** command.

- **Display accuracy**

Controls the number of significant digits displayed in the grid line markers. Set this property to the number of places you want displayed to the right of the decimal point. Enter a value between **0** (for integer numbers) and **6**. The default is **4**.

- **Resolution**

Permits plotting of only a subset of the collected data points. This is useful for increasing rendering speed. A divisor value of n causes only every n th point to be plotted. Thus, if **1000** points are being collected and the **Resolution** is **5**, then **200** points are displayed. Of course, you may not want to reduce resolution if your data can contain glitches or other high-frequency phenomena. The default is **1**.

- **Grid spacing**

Specifies the minimum distance (in pixels) allowed between the grid lines. Increasing this number decreases the number of grid lines, decreasing this number increases the number of grid lines. Enter a value between **5** and **500**. The default is **100**.

Additional options for **Windows** hosts only are:

- **Snap measures to signals**

Controls whether or not measures are snapped to the signal lines on the grid. Measures are described in [On-grid Measurements](#), p.78. The default is **True**.

- **Maximum snap distance**

Controls how far away (in pixels) you can start a measure and still have it snap to the plot line. Measures are described in [On-grid Measurements](#), p.78. The default is 20.

- **Background image**

Allows you to specify special images to be displayed in the graphing area, with the graph itself superimposed on top. The default is none.

Additional options for **UNIX** hosts only are:

- **Aspect Lock**

When selected, it causes the aspect ratio of the plot window to be maintained when you zoom in or out on the plot.

To modify any value, just type directly into the text field. Any changes you make in this window have no effect on any other open **Plot XY** windows.

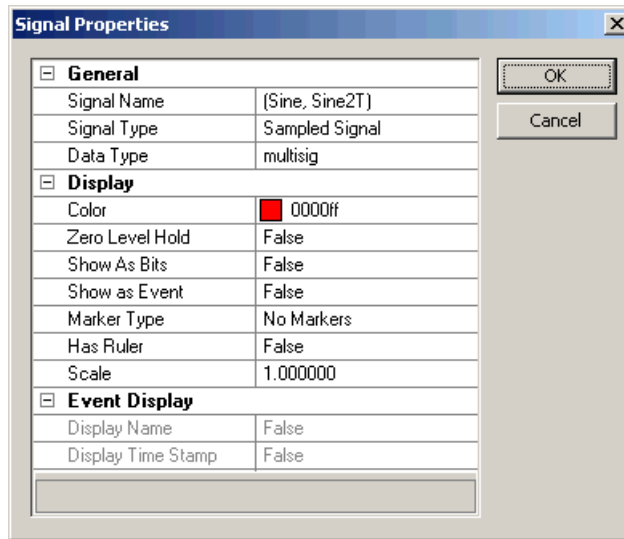
Any changes you make in this window have no effect on any other open **Plot XY** windows.

To change the defaults used when new **Plot XY** windows are created, use the **File > Preferences** menu command (or the **Preferences** toolbar button - see [Main Toolbar](#), p.40), described in [3.3.12 Preferences](#), p.53.

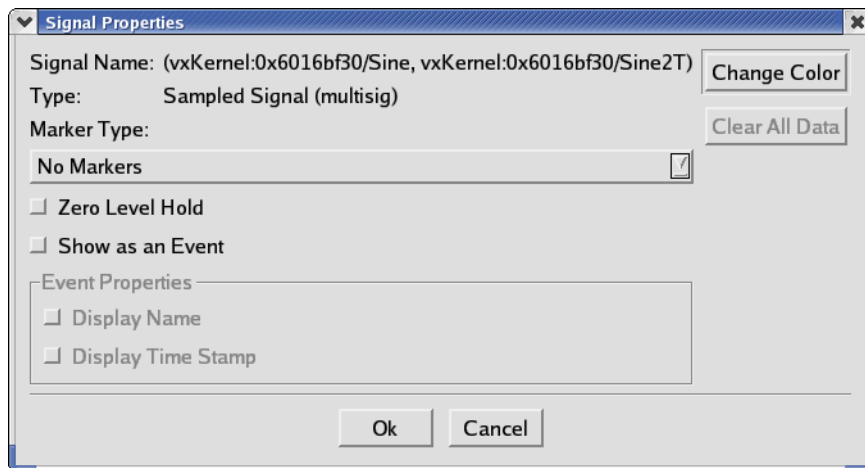
8.4 Signal Properties Dialog Box

Characteristics of the line used to plot each signal in this window can be configured using the **Signal Properties** dialog box.

Windows Host



UNIX Host



The dialog box can be opened in either of the following ways:

- Right-click a signal in the **Signals** tab view (see [Signals Bar](#), p.149).
- Right-click a signal trace in the data-display area.

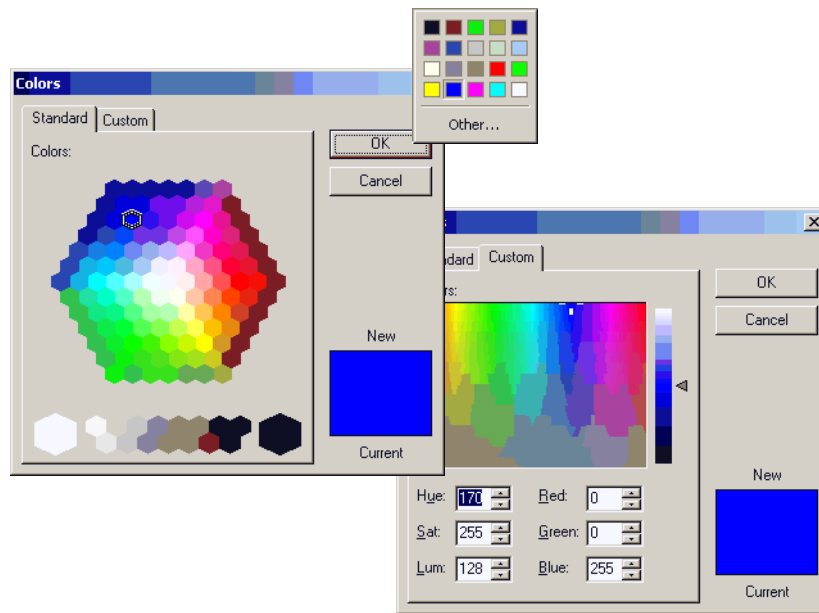
Signal line properties options in both Windows and UNIX hosts are:

- **Signal Name and Type**

These first two lines are the name of the signal (as shown in the signals tree), and its type.

- **Color**

Clicking on the current value opens a small color palette with a basic color selection from which to choose.



If you would like a color that is not on this color palette, you can click **Other** to open the **Colors** dialog box where you can select a new color from the larger selection in the **Standard** tab view, or you can create an entirely new color using the **Custom** tab view.

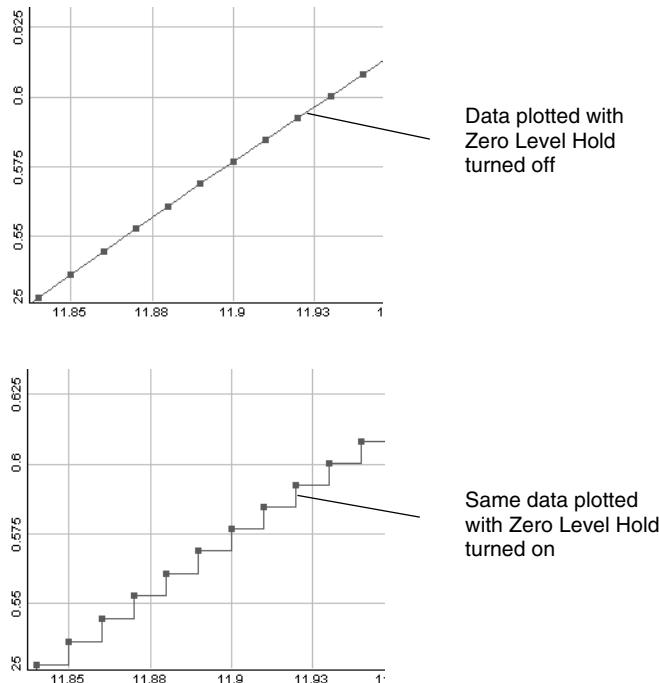


NOTE: An alternative method for changing colors of plotted lines on the graph is given in the discussion of colors preferences in [Colors View](#), p.56.

- **Zero Level Hold**

Select **True** from the drop-down menu to hold the value of the signal until the next sample arrives. The default is **False**.

To demonstrate this feature, consider the following sampling of a sine wave at a rate of 100Hz.



In the upper view in this figure, with **Zero Level Hold** turned off (= **False**), the data points are connected as usual with straight lines, even though there is no information about the actual values between data points. In the lower view, the same data, but with the **Zero Level Hold** feature turned on (= **True**), shows how the line plotted between the same data points is now a straight horizontal line of the same value as the previous data point, until the next data point comes in, at which time the plot line goes vertical up to that value.




- **Show as Event**

Select **True** from the drop-down menu to cause samples to be marked with vertical lines instead of connecting the individual samples with lines. The

default is **False**. For more information, see [8.5 Setting New Plot XY Window Preferences](#), p.160.

- **Marker Type**

This drop-down list allows you to select a different symbol (or marker) to be plotted for each sample of this signal. The choices are:

- **No Markers** — No markers showing
- **Square** — "  "
- **Plus** — "  "
- **Diamond** — "  "

The default is **No Markers**.

- **Display Name**

Causes the name of the event (**event ID**) to be displayed along side the vertical marker.

- **Display Time Stamp**

Causes the timestamp to be displayed along side the vertical marker.

Additional options for **Windows** hosts only are:

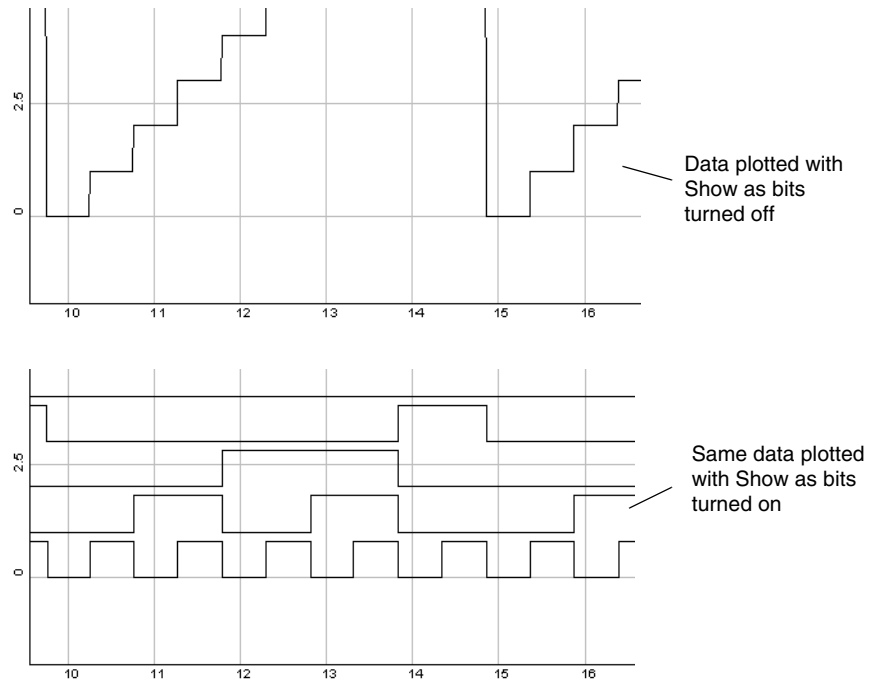
- **Data Type**

This is the C++ data type of this signal (program variable) For a list of allowable types, see [Table 16-1](#).

- **Show As Bits**

Select **True** from the drop-down menu to cause a signal (program variable) being plotted to be represented as a sequence of bits, each bit with its off and on (**0** and **1**) values indicated. The bits are rendered individually by discrete horizontal lines distributed up the Y axis from bit position 0 starting at the bottom. The default is **False**.

To show this feature, a **Derived Signal**, created using a single 8-bit integer, is incremented through values from 0 to 15 and back to 0 again at the rate of two increments per second, and then repeated continuously until stopped, resulting in the plot shown here.



The top graph shows the value of the variable with **Show as bits** turned off (= **False**), and the bottom graph shows the same data with **Show as bits** turned on (= **True**). You can easily observe the pattern of the individual bits (horizontal lines) making up the 16-bit integer, showing their 0 and 1 positions with respect to time.

This feature works equally well with all signal types, but the resulting bit patterns may be more difficult to decipher depending on the complexity of the signal.

▪ **Has Ruler**

This true/false option displays a separate ruler with colored numbers matching the color of the signal on the left (Y) axis. If you have, say, 3 signals with rulers on, there are three separate color-coded rulers on the left plot boundary. The default is **False**.

- **Scale**

Each plotted value is scaled (multiplied) by the value you enter. The default is 1.000000.

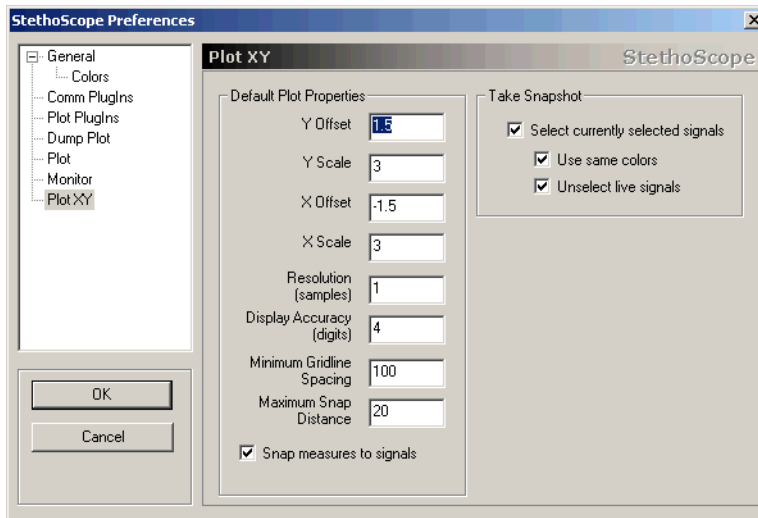
Additional options for **UNIX** hosts only are:

- **Clear All Data (button)**

Resets all the signal properties to Data Monitor default values.

8.5 Setting New Plot XY Window Preferences

The **Preferences** dialog box allows you to set default parameters for Data Monitor when it starts, and for new data-display windows when they are created (whereas the properties described in this chapter apply only to currently open windows). The **Plot XY Preferences** view is where you change the default values used when **Plot XY** windows are created. It also allows you to control what happens when a snapshot is taken.



To modify these preferences, open the **Preferences** dialog box using the **File > Preferences** menu command (or the **Preferences** toolbar button - see [Main Toolbar](#), p.40), then click **Plot XY** in the left panel to make the following parameters available for configuration.

Default Plot Properties Panel

- **Y Offset**

Controls the unit value for the top coordinate on the plot. When first started, it is **1.5**, and the Y value of **1.5** is the top value. This value changes automatically if you use the **Zoom to Fit** command.

- **Y Scale**

Controls the height of the plot grid in units. For example, when first started, Y Range defaults to **3** and the plot displays the Y-axis from **1.5** to **-1.5**.

- **X Offset**

Controls the unit value for the left-most coordinate on X axis of the plot. When first started, it defaults to **0**. This value changes automatically if you use the **Zoom to Fit** command.

- **X Scale**

Controls the width of the plot grid in units. For example, when first started, X Range defaults to **20**, and the plot displays the X-axis from **0** to **20**.

- **Resolution (samples)**

Permits plotting of only a subset of the collected data points. This is useful for increasing rendering speed. A divisor value of n causes only every n th point to be plotted. Thus, if **1000** points are being collected and the **Resolution** is **5**, then **200** points are displayed. Of course, you may not want to reduce resolution if your data can contain glitches or high-frequency phenomena. The default is **1**.

- **Display Accuracy (digits)**

Controls the accuracy of the grid line markers. Set this property to the number of places to the right of the decimal point to use in the grid markers. Enter a value between **0** (for whole numbers) and **6**. The default is **4**.

- **Minimum Gridline Spacing**

Specifies the minimum distance (in pixels) allowed between the grid lines. Increasing this number decreases the number of grid lines, decreasing this number increases the number of grid lines. Enter a value between **5** and **500**. The default is **100**.

- **Minimum Snap Distance**

Specifies the minimum distance (in pixels) from a grid line that will cause a plot point to be snapped to the grid line. The default is **20**.

- **Snap measures to signals**

If this box is selected, the end point of a measurement line is snapped to the signal plot line, provided it is within the **Minimum Snap Distance** from the plot line when you release the mouse button. Otherwise the measurement only made to the point where you release the mouse button. The default is **selected**.

Take Snapshot Panel

Unlike all other preferences, which only impact data-display windows you may create later, these preferences take effect immediately. For more information, see [11. Working with Snapshots](#).

The following functions can be selected by setting their check boxes:

- **Select currently selected signals**

If selected, the same signals currently selected in the live buffer are also selected in the snapshot.

- **Use same colors**

If selected, the same colors are assigned to the signals in the snapshot. If you want the snapshot to use different colors for each signal than are used for live data, clear this check box.

- **Unselect live signals**

If selected, the live signals will become unselected when the snapshot is taken.

9

The Dump Plot Window

[9.1 Introduction 163](#)

[9.2 Dump Plot Window Tour 164](#)

[9.3 Setting New Dump Plot Window Preferences 171](#)

9.1 Introduction

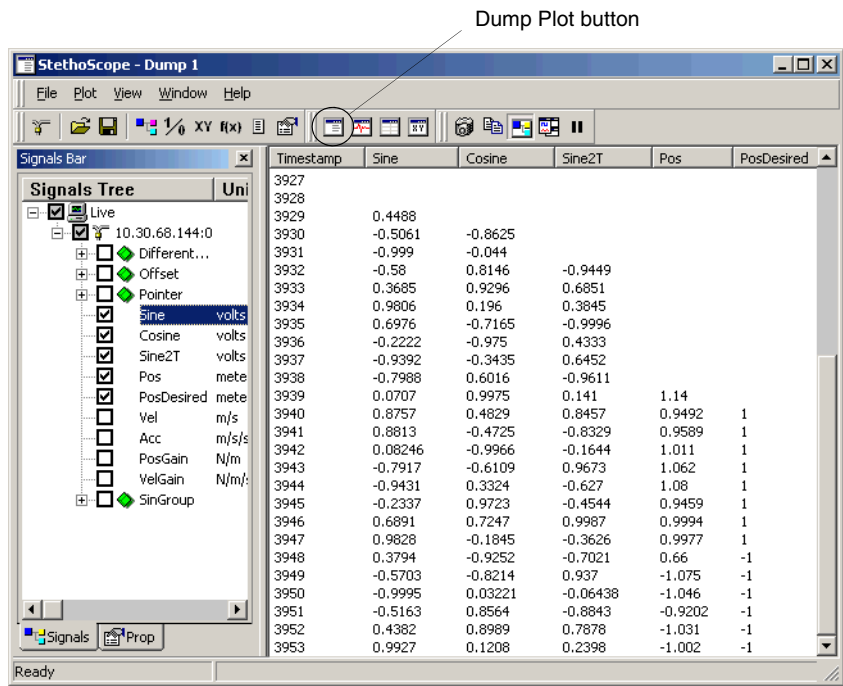
The **Dump Plot** window displays the value of each signal at each sampling, in a tabular format, scrolling with time. Like the **Plot** window, a **Dump Plot** window can display both live and snapshot data. You can have multiple **Dump Plot** windows open at the same time, and each can display different signals or snapshots.

This chapter describes the characteristics and use of the **Dump Plot** window in detail.

Before using a **Dump Plot** window, it may help to understand how and when data is collected from the target. For insights on this issue, see [5. Triggering](#).

9.2 Dump Plot Window Tour

To open a **Dump Plot** window, use the **File > Plots > Dump Plot** menu command (or the **Dump Plot** toolbar button - see [Plots Toolbar](#), p.41). The following is a **Dump Plot** window with example data.



The left-most data column is always the **TimeStamp**, calculated by multiplying the sample period by the position of each sample in the data buffer. TimeStamp is reset to zero for each collected data set. The other columns in this table are the values, at the time increments, for each signal selected in the **Signals Tree** to the left. This table is dynamically updated as data collection progresses.

Displaying Signal Parameters

To display signal parameters, do the following:

1. If you do not already have a **Signals Bar** open in your **Dump Plot** window, open one using the **View > Signals Bar** menu command (or the **Signals Bar** toolbar button - see [Plot Window Toolbar](#), p.41). Make sure the **Active Signals** tab is selected in the Signals Bar so that a **Signals Tree** is displayed.
2. Use the **Signals Tree** to select which signals you want to display in the table. Each selected signal appears as a column in the table. For details on using Signals Trees, see [4. Using the Signal Manager](#).

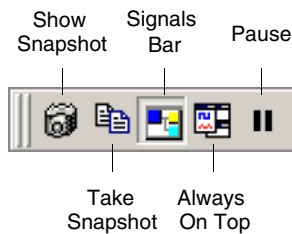
Most of the remaining functionality in the **Dump Plot** window is provided through the toolbars and menus. For common data-display window feature descriptions, refer to the following sections:

- [3.2 Toolbars](#), p.40
- [Plot Menu Item \(Windows Hosts Only\)](#), p.69
- [View Menu Item](#), p.70

Toolbar

In a default **Dump Plot** window, the first two toolbars are identical to the toolbars shown and with the menu items as described in Sections [Main Toolbar](#), p.40 and [Plots Toolbar](#), p.41. But the right-most toolbar is specific to the **Dump Plot** window. All the toolbars are dockable (as well as the menu bar), which means you can move them to other locations, on or off the window, simply by dragging them. Each of the toolbars can be independently displayed or hidden using the **View** menu item (see [View Menu Item](#), p.166).

The **Dump Plot** window toolbar is a subset of the one shown and described in detail in [Plot Window Toolbar](#), p.41.

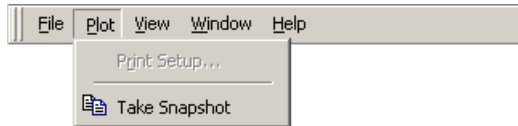


Menu Bar

Some of the menu bar items are discussed in detail in [3.3 File Menu Item](#), p.43, where it is mentioned that the **File**, **Window**, and **Help** menu items are consistently the same across all data-display windows. The **Plot** and **View** menu items, however, contain some commands that are unique to the **Dump Plot** window. Even though partially redundant, these menu items are described in detail in the following sections.

Plot Menu Item (Windows Hosts Only)

In a Windows host, the **Plot** menu item contains commands for working with the plots in the **Dump Plot** window.



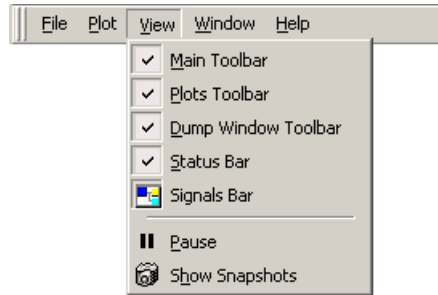
The **Plot** menu options for the **Dump Plot** window are:

- **Print Setup**
Allows you to select printer parameters and characteristics before printing (not enabled at this time).
- **Take Snapshot**
Saves a copy of all the active signals (not just the selected signals), for all connected targets. For more information, see [11.2 Utilizing Snapshots](#), p.185.

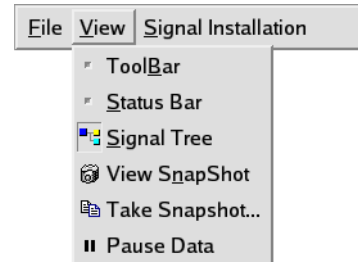
View Menu Item

The **View** menu item contains options for working with the plots in the **Plot** window.

Windows Host



UNIX Host



The following menu options are available for both Windows and UNIX hosts:

- **Main Toolbar (Toolbar in UNIX hosts)**
Controls whether the toolbar representing a selection of items from the **File** menu is displayed on the **Dump Plot** window toolbar. For more information, see [3.2 Toolbars](#), p.40.
- **Status Bar**
Controls whether the status line along the bottom of the window is visible. For information, see [3.7 Status Bar](#), p.78.
- **Signals Bar (Signal Tree in UNIX hosts)**
Controls whether a **Signals Bar** panel appears in the window. The **Signals Bar** (see the figure in [9.2 Dump Plot Window Tour](#), p.164) includes tabs for **Signals** and **Properties**. For more information, see [Signals Bar](#), p.168.
- **Pause (Pause Data in UNIX hosts)**
Stops updating the table in the **Dump Plot** window. It does not stop data collection, but merely stops new data from appearing in the **Dump Plot** table. To resume normal display, unselect the button.
- **Show Snapshots (View Snapshot in UNIX hosts)**
Controls what is displayed in the **Signals Tree**. When selected, only snapshots appear in the **Dump Plot** window **Signals Tree**. When unselected, only the live data buffer appears. For more information, see [11.2 Utilizing Snapshots](#), p.185.

Additional options for **Windows** hosts only are:

- **Plots Toolbar**

Controls whether the toolbar representing a selection of items from the **File > Plot** menu command is displayed on Data Monitor's toolbar. For more information, see [3.2 Toolbars](#), p.40.

- **Dump Window Toolbar**

Controls whether the toolbar used within the **Dump Plot** data-display window is visible. The buttons represent items from the **View** menu for the **Dump Plot** data-display window. For more information, see [Toolbar](#), p.165.

Signals Bar

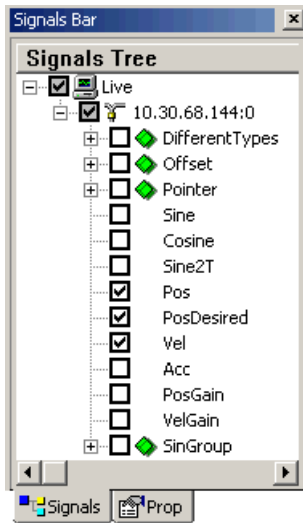
The **Signals Bar** is a sub-window in the Data Monitor GUI that allows you to view and configure information that affects how signals appear in the data-display window. If a **Signals Bar** window is not displayed, you can open one using the **View > Signals Bar** menu command (or the **Signals Bar** toolbar button - see [Plot Window Toolbar](#), p.41).

The **Signals Bar** contains two tab views:

- [Signals Tab View](#)
- [Properties Tab View](#)

Signals Tab View

The **Signals Tree**, in the **Signals** tab view of the **Signals** bar, displays all the signals available to be plotted.



They are shown in an expandable tree structure containing signals that have been installed by the **Signal Manager** (see [4. Using the Signal Manager](#)). Signals in the table are color-coded to the signals selected in the Signals Tree. The signals are shown in an expandable tree structure, containing signals that have been installed by the **Signal Manager** (see [4. Using the Signal Manager](#)). Signals in the table are color-coded to the signals selected in the tree. The Signals Tree is displayed by default when you open a **Dump Plot** window, but it may be re-displayed at any time by selecting the **Signals** tab at the bottom of the window.

You can locate any signal that has been installed and add it to the graph of signal traces using the **Signals Tree**. Each node of the tree (as well as each individual signal) has an associated check box. Selecting the check box for an individual signal adds that signal to the graph, or, conversely, clearing a check box removes that signal from the graph. If you select a node check box, all the signals belonging to that node (including any nodes below it) are selected at once and those signals are added to the graph. Conversely, clearing a node check box removes all signals, and nodes, below that node from the graph with a single click.

If a node check box is selected, but has a grey fill, it indicates that some, but not all, of the signals belonging to that node are selected and displayed on the graph. You may have to scroll down to see which ones are selected.

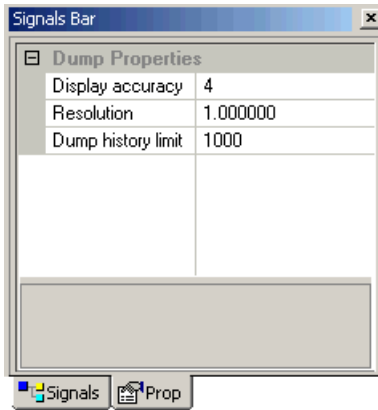
The signals tree is expanded or collapsed using the icon to the left of each node check box as follows:

- "+" = **collapsed**; select to expand down to the next node(s).
- "-" = **expanded**; select to collapse up to this node.

The **Units** column in the **Signals** tab view shows the physical measurement units of each signal. You may have to move column or window boundaries around in order to see the Units column.

Properties Tab View

You can view and change the properties for the currently open **Dump Plot** window using the **Properties** tab in the **Signals Bar**.



The display window environment properties shown in this tab view are grouped under the **Dump Properties** panel. They are:

- **Display accuracy**
Controls the number of significant digits displayed in the table. Set this property to the number of places you want displayed to the right of the decimal point. Enter a value between 0 (for integer numbers) and 6. The default is 4.
- **Resolution**
Controls how often (in seconds) to refresh values in the **Dump Plot** table. The default is 1.000000.

- **Dump history limit**

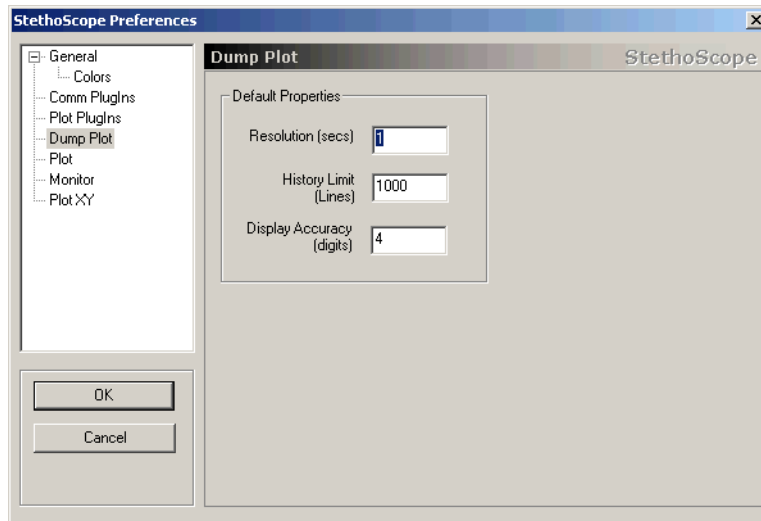
Controls how many lines of historical data to maintain in the **Dump Plot** table, 0 = display all. The default is **1000** (500 in a UNIX host).

To modify any value, just type directly into the text field. Any changes you make in this window have no effect on any other open **Dump Plot** windows.

9.3 Setting New Dump Plot Window Preferences

The **Preferences** dialog box allows you to set default parameters for new **Dump Plot** windows when they are created, whereas the properties described in [Properties Tab View](#), p.170 apply only to currently open windows.

The **Dump Plot** view of the **Data Monitor Preferences** dialog box (see [3.3.12 Preferences](#), p.53) allows you to change these default values.



These preference changes have no effect on **Mini Dump Plot** windows (within **Plot** windows), or on already open **Dump Plot** windows. They will only affect **Dump Plot** windows opened after the preferences are modified.

To modify these preferences, open the **Preferences** dialog box using the **File > Preferences** menu command (or the **Preferences** toolbar button - see [Main Toolbar](#), p.40), then click **Dump Plot** in the left panel to set the following parameters:

Default Properties Panel

- **Resolution (secs)**

Controls how often, in seconds, to refresh the values in the **Dump Plot** window. The default is **1**.

- **History Limit (Lines)**

Controls how many lines of historical data to maintain in the v window. The default is **1000**.

- **Display Accuracy (digits)**

Controls the accuracy of values displayed in the table. Set this property to the number of places to the right of the decimal point to use. Enter a value between **0** (for whole numbers) and **6**. The default is **4**.

10

The Monitor Window

- 10.1 Introduction 173
- 10.2 Monitor Window Tour 174
- 10.3 Writing Data to the Target 181
- 10.4 Setting New Monitor Window Preferences 182

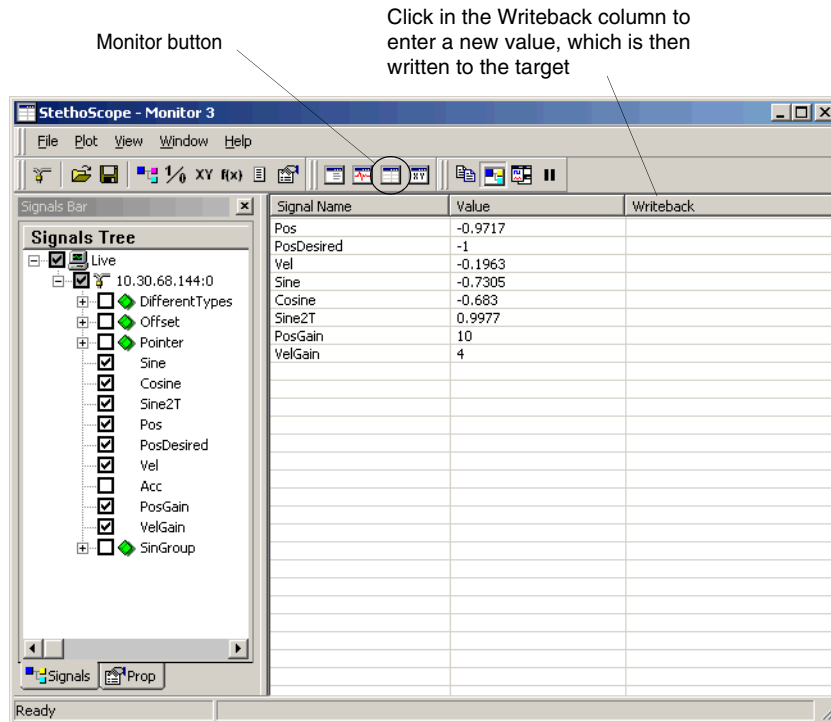
10.1 Introduction

The **Monitor** window differs from the **Dump Plot** window in that the **Dump Plot** window displays a scrolling list of signal values as they change over time, whereas the **Monitor** window only shows you the current value of each monitored signal. The **Monitor** window can also be used to modify the value of a signal on the target.

This chapter describes the characteristics and use of the **Monitor** window in detail.

10.2 Monitor Window Tour

To open a **Monitor** window, use the **File > Plots > Monitor** menu command (or the **Monitor** toolbar button - see [Plots Toolbar](#), p.41). Shown here is an example **Monitor** window with data from the demonstration program.



You can select the signals you want to monitor by using the **Signals** tab in the **Signals Bar**. Each selected signal appears as a row in the **Monitor** window table with the following column descriptions.

- **Signal Name**
The first column displays the
- **Value**
The most recent value for each signal.

- **Writeback**

Used to write new values back to the target. This column only appears when the **Writeback** property in the **Properties** tab view is set to **True** (see [Properties Tab View](#), p.180, and [10.3 Writing Data to the Target](#), p.181).

Displaying Signal Parameters

To display signal parameters, follow these steps:

1. If you do not already have a **Signals Bar** in your **Monitor** window, use the **View > Signals Bar** menu command (or the **Signals Bar** toolbar button - see [Plot Window Toolbar](#), p.41). Make sure the **Signals** tab is selected in the Signals Bar, so that a **Signals Tree** is displayed.
2. Use the Signals Tree to select which signals you want to display in the table.
For details on using Signals Trees, see [4. Using the Signal Manager](#).

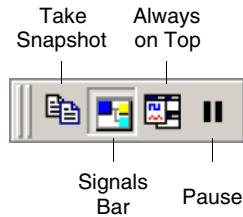
Most of the remaining functionality in the Monitor window is provided through the toolbars and menus. For more information on common data-display window features, refer to the following sections:

- [3.2 Toolbars](#), p.40
- [3.3 File Menu Item](#), p.43

Toolbar

In a default **Monitor** window, the first two toolbars are identical to the toolbars shown and with the menu items as described in Sections [Main Toolbar](#), p.40 and [Plots Toolbar](#), p.41. But the right-most toolbar is specific to the **Monitor** window. All the toolbars are dockable (as well as the menu bar). Each of the toolbars can be independently displayed or hidden using the **View** menu item (see [View Menu Item](#), p.177).

The **Monitor** window toolbar is a subset of the one shown in [Plot Window Toolbar](#), p.41, where the icons are described in detail.

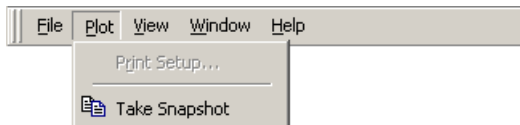


Menu Bar

Some of the **Menu Bar** items are discussed in detail in [3.3 File Menu Item](#), p.43, where it is mentioned that the **File**, **Window**, and **Help** menu items are consistently the same across all data-display windows. For the **Monitor** window, however, the **View** menu item contains some commands that are unique to the **Monitor** window. Even though partially redundant, these menu items are described in detail in the following sections.

Plot Menu Item (Windows Hosts Only)

In a Windows host, the **Plot** menu item contains commands for working with data in the **Monitor** window.



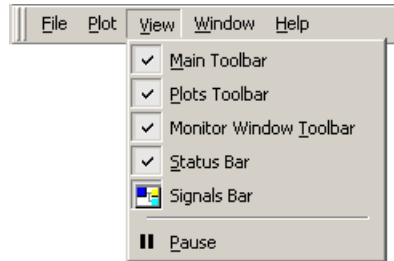
The **Plot** menu options in the **Monitor** window are:

- **Print Setup**
Allows you to select printer parameters and characteristics before printing.
- **Take Snapshot**
Saves a copy of all the active signals (not just the selected signals), for all connected targets. For more information, see [11.2 Utilizing Snapshots](#), p.185.

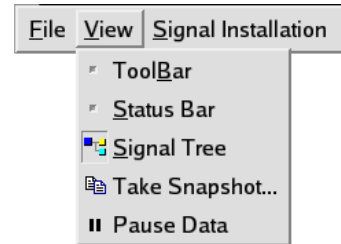
View Menu Item

The **View** menu item contains commands primarily used for choosing which elements of the **Monitor** window to display.

Windows Host



UNIX Host



10

The following menu options are available in both Windows and UNIX hosts:

- **Main Toolbar (Toolbar in UNIX hosts)**

Controls whether the toolbar representing a selection of items from the **File** menu is displayed on the Data Monitor toolbar. For more information, see [3.2 Toolbars](#), p.40.

- **Status Bar**

Controls whether the status line along the bottom of the window is visible. For more information, see [3.7 Status Bar](#), p.78.

- **Signals Bar (Signal Tree in UNIX hosts)**

Controls whether a **Signals Bar** panel appears in the window. The **Signals Bar** (see [Signals Tab View](#), p.178) includes tabs for **Signals** and **Properties**. For more information, see [Signals Bar](#), p.178.

- **Pause**

Stops updates to the table in the **Monitor** window. It does not stop data collection, but merely stops new data from appearing in the table. To resume normal display, unselect the button.

Additional options for **Windows** hosts only are:

- **Plots Toolbar**

Controls whether the toolbar representing a selection of items from the **File > Plot** menu command is displayed on the Data Monitor toolbar. For more information, see [3.2 Toolbars](#), p.40.

- **Monitor Window Toolbar**

Controls whether the toolbar used within a specific data-display window is visible. The buttons represent items from the **Plot** and **View** menus for the specified data-display window. For more information, see [Toolbar](#), p.175.

Additional options for **UNIX** hosts only are:

- **Take Snapshot**

Saves a copy of all the active signals. You can display the snapshot in the **Plot XY** window along with real-time data. For more information, see [11.2 Utilizing Snapshots](#), p.185

Signals Bar

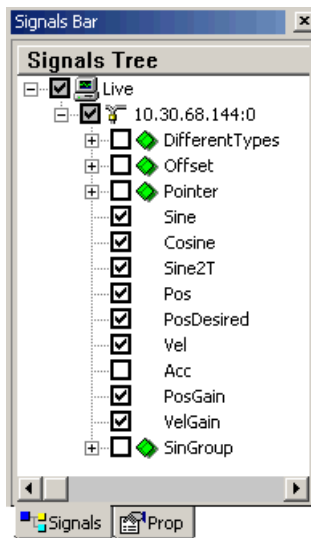
The **Signals Bar** is a sub-window in the Data Monitor GUI that allows you to view and configure information that affects what signals appear in the data-display table. If a **Signals Bar** panel is not displayed, you can open one using the **View > Signals Bar** menu option (or the **Signals Bar** button).

The **Signals Bar** contains the following tab views:

- [Signals Tab View](#)
- [Properties Tab View](#)

Signals Tab View

The **Signals Tree** in the **Signals** tab view displays all the signals available to be plotted.



The Signals Tree is displayed by default when you open a Monitor window, but it may be re-displayed at any time by clicking the **Signals** tab at the bottom of the sub-window.

The Signals Tree allows you to select any signal that has been installed and add it to the list of signal values. Each node of the tree (as well as each signal) has a check box. Selecting the check box for an individual signal adds that signal to the graph, or, conversely, clearing a check box removes that signal from the graph. If you select a node check box, all the signals belonging to that node (including their nodes below it) are selected at once and those signals are added to the graph. Conversely, clearing a node check box removes all signals below that node from the graph with the single click.

If a node check box is selected, but has a grey fill, it indicates that some, but not all, of the signals belonging to that node are selected and displayed on the graph. You may have to scroll down to see which ones are selected.

The Signals Tree is expanded or collapsed using the icon to the left of each node check box as follows:

- "+" = **collapsed**; select to expand down to the next node(s).
- "-" = **expanded**; select to collapse up to this node.

The **Units** column in the **Signals** tab view shows the physical measurement units of each signal.

The **Signals Tree** is displayed by default when you open a **Monitor** window, but it may be re-displayed at any time by clicking the **Active Signals** tab at the bottom of the sub-window.

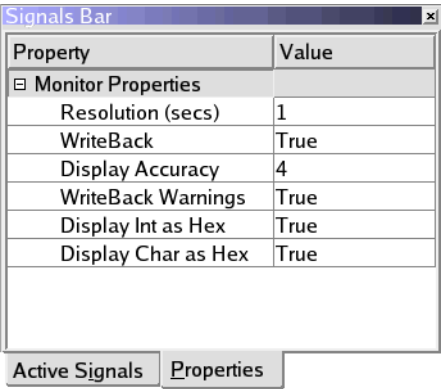
Properties Tab View

You can view and change the properties for the currently open Monitor window by clicking the **Prop(erties)** tab in the Signals Bar.

Windows Host



UNIX Host



The following menu options are available in both Windows and UNIX hosts:

- **Display accuracy**
Controls the number of significant digits displayed in the table. Set this property to the number of places you want displayed to the right of the decimal point. Enter a value between **0** (for integer numbers) and **6**. The default is **4**.
- **Monitor resolution**
Controls how often, in seconds, to refresh the values in the Monitor window. The default is **1.000000**.
- **Allow writeback**
Select **True** to create a **Writeback** column for writing modified signal values back out to the target. The default is **False**.

Additional options for UNIX hosts only are:

- **Writeback Warnings**

Controls whether or not a warning is displayed before each WriteBack attempt. The default is **True**.

- **Display Int as Hex**

Controls whether or not to display integer values in hexadecimal. The default is **True**.

- **Display Char as Hex**

Controls whether or not to display char variables in hexadecimal. The default is **True**.

To modify any value, just type directly into the text field. Any changes you make in this window have no effect on any other open Monitor windows.

To change the defaults used when new Monitor windows are created, see [10.4 Setting New Monitor Window Preferences](#), p.182.

10

10.3 Writing Data to the Target

Data Monitor provides you with **Writeback**, a very powerful feature with which you can change the values of variables on the target as your program runs.



WARNING: Modifying values in a running program is powerful, but can be dangerous. Be very careful. You should realize that the wrong value may be written by accident, either by you or by the Data Monitor program. This could be caused by a mistyped entry, an error in determining the correct address or type for a variable, or a bug in the Data Monitor program itself.

THIS FACILITY SHOULD NOT BE USED TO CONTROL SAFETY-CRITICAL SYSTEMS. Use it at your own risk!

A message to this effect will be displayed when a value is written. You can disable this message using the check box in the Warning dialog box that opens.

Using Writeback

To enable and use the writeback feature, follow these steps:

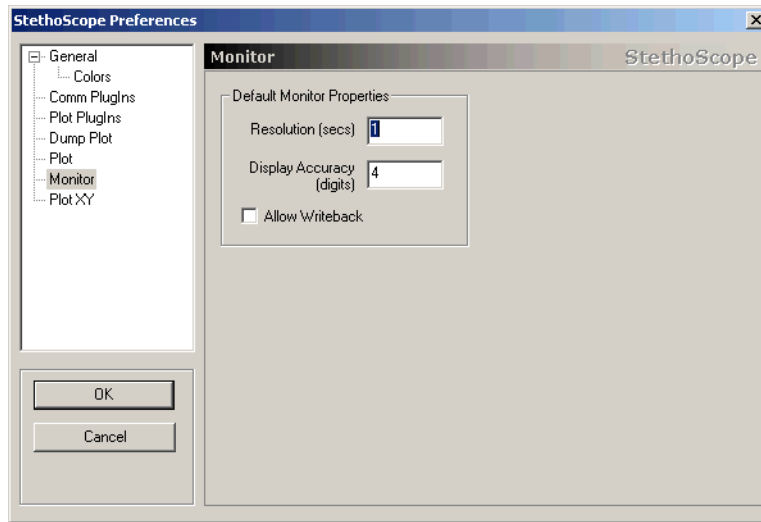
1. Make sure Writeback is set to **True** in the **Properties** tab view of the **Signals Bar** (see [Properties Tab View](#), p. 180). This causes the Writeback column to appear in the table.
2. Enter a value in the Writeback column for the desired signal at any time. It can be a number or the name of another signal. In the latter case, the last available value of the named signal is used.
3. With the mouse pointer inside the Writeback field, press the **Enter** key to write the value to the target. (The warning message above is displayed first.)

Writing a value with the Monitor window changes the value of the variable in the target memory. The change occurs asynchronously (when the data arrives).

10.4 Setting New Monitor Window Preferences

The **Preferences** dialog box allows you to set default parameters for new **Monitor** windows when they are created (whereas the properties described in this chapter apply only to currently open windows).

The **Monitor Preferences** panel allows you to change these default values.



These preferences have no effect on **Mini-Monitor** windows within Plot windows, or already open Monitor windows. (To change these values on Monitor windows you have already created, see [Properties Tab View](#), p.180.)

To modify these preferences, open the **Data Monitor Preferences** dialog box with the **File > Preferences** menu command (or the **Preferences** toolbar button; see [Main Toolbar](#), p.40), then click **Monitor** in the left panel to set the following options:

- **Resolution (secs)**
Controls how often (in seconds) to refresh the values in the **Monitor** window. The default is 1.
- **Display Accuracy (digits)**
Controls the accuracy of values displayed in the table. Set this property to the number of places to the right of the decimal point to use. Enter a value between 0 (for whole numbers) and 6. The default is 4.
- **Allow Writeback**
Check box controls whether or not you can use the **Monitor** window to write modified signal values back out to the target. Opens a **Writeback** column in the table if selected. The default is **unselected**.

11

Working with Snapshots

[11.1 Introduction 185](#)

[11.2 Utilizing Snapshots 185](#)

11.1 Introduction

This chapter describes the taking and processing of snapshots in detail.

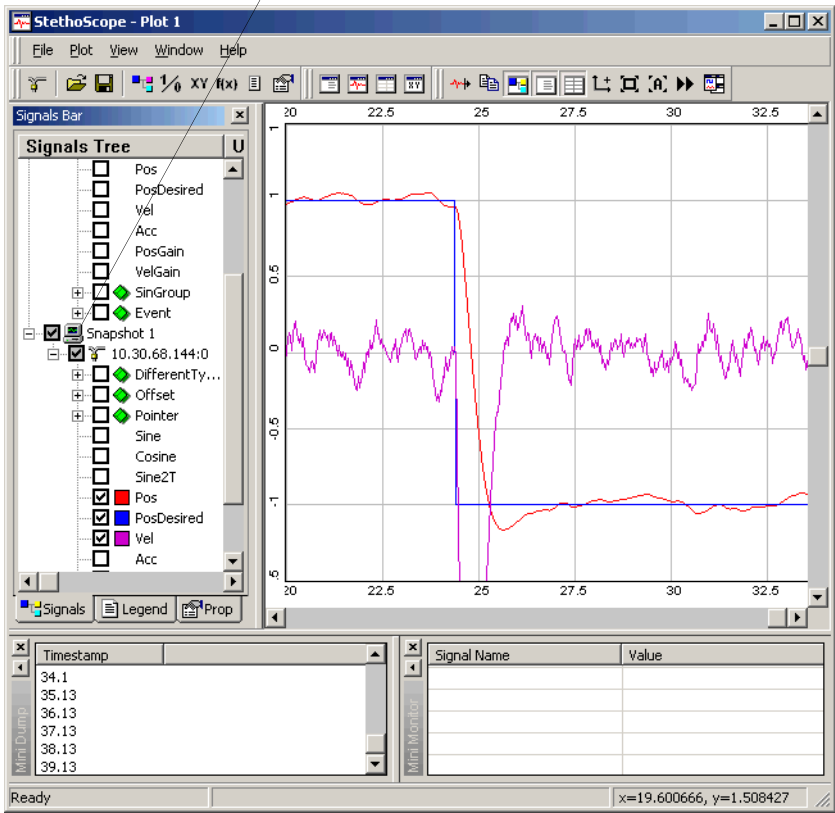
11.2 Utilizing Snapshots

For all connected targets, a snapshot saves all the data collected for all active signals since the current Data Monitor session began. You can display snapshots in the **Plot** and **Plot XY** windows, along with live data and other snapshots. This makes it easy to compare test runs. You can save snapshots in four different formats for use in other applications. You can take snapshots, save snapshots to disk, and load snapshots from disk.

Taking Snapshots

A snapshot is not directly labeled as such, but rather it shows up selected at the bottom of the Signals tree. An example of a snapshot is shown here.


When the snapshot is inserted at the bottom of the Signals Tree, it is selected and the live buffer is unselected



The Snapshot Process

When you connect to a target, Data Monitor collects active signals and stores them in the live data buffer. Taking a snapshot creates a copy of that live buffer. And since the live buffer includes all **active** signals, the snapshot buffer also includes all active signals, even though only **selected** signals actually appear in the plot.

There are multiple ways to take a snapshot, as follows:

- Using the **Take Snapshot** button () on the toolbar.
- Using the **Plot > Take Snapshot** menu command.
- Using the **File > Save Snapshot** menu command, with Snapshot type set to **Live** and the **Write** field set to **Immediately, in the middle of this cycle**.
- Using the **Triggering** dialog box to configure and arm a trigger and set automatic **Snapshot**.

When you take a snapshot, all of the live data received is copied into a temporary snapshot buffer. No event data other than the collected event identifiers is stored.

What happens next depends on the **Take Snapshot** settings in the **Plot Preferences** panel of the **Preferences** dialog box (see [7.6 Setting New Plot Window Preferences](#), p.134), but the following describes the default behavior.

In the **Signals Tree**, the live buffer becomes deselected and the snapshot is added at the bottom of the Signals Tree and becomes selected.

The plot grid area switches from displaying the live buffer to displaying the newly saved snapshot. While the live buffer is no longer shown as selected in the Signals Tree, real-time data is still being collected in the live buffer.



NOTE: When the snapshot is first displayed on the grid, it shows **all active signals**. Nothing is wrong with the display even though it may look quite cluttered. You likely want to select only the signals you want to view in the same manner used for selecting live signals.

You can display the live buffer, plus multiple snapshots, all at the same time. This makes it easy to compare previous runs with real-time runs.

Snapshots are automatically given temporary storage names such as **snapshot1**, **snapshot2**, and so forth. You can change the names when you save the snapshots to disk.

Snapshots are in temporary storage until, and unless, you save them to disk, as described in the next paragraph. Any snapshots you do not save before exiting Data Monitor will be lost.

Saving Snapshots

While the **Take Snapshot** command creates a copy of real-time data in a temporary buffer, the **Save Snapshot** command stores that buffered data in your file system.

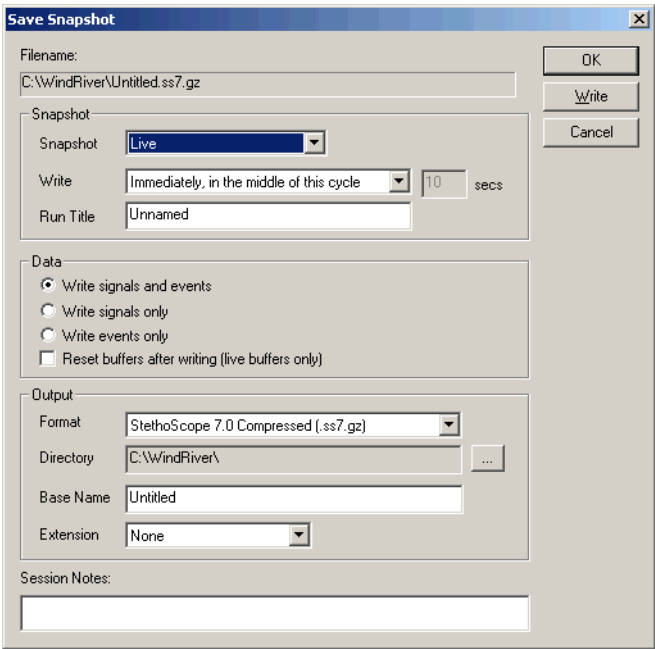


NOTE: The snapshot data in the temporary buffer will be lost when you exit Data Monitor if you do not save it to a file first.

Save Snapshot Dialog Box

The **File > Save Snapshot** menu command (or the **Save** button) opens the **Save Snapshot** dialog box.

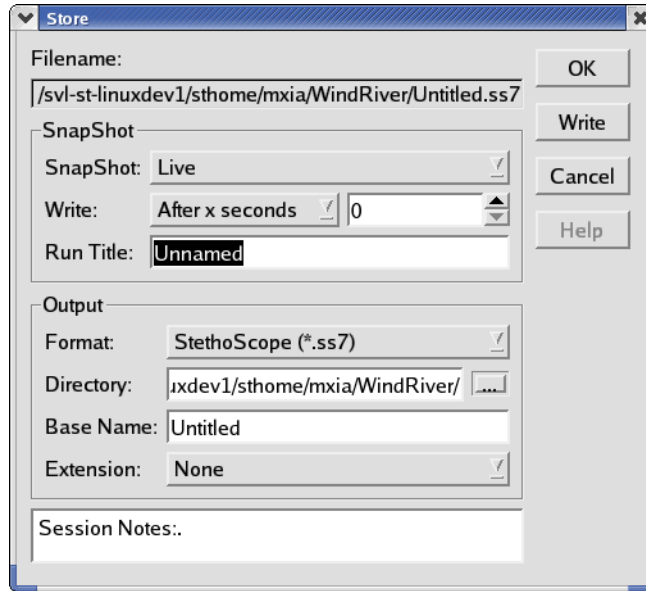
Windows Host



The **Save Snapshot** dialog box is shown with the following fields and controls:

- Filename:** C:\WindRiver\Untitled.ss7.gz
- Snapshot:** Live (dropdown menu)
- Write:** Immediately, in the middle of this cycle (dropdown menu) 10 secs
- Run Title:** Unnamed
- Data:**
 - ☒ Write signals and events
 - ☐ Write signals only
 - ☐ Write events only
 - ☐ Reset buffers after writing (live buffers only)
- Output:**
 - Format:** StethoScope 7.0 Compressed (.ss7.gz) (dropdown menu)
 - Directory:** C:\WindRiver\ (text field with browse button)
 - Base Name:** Untitled (text field)
 - Extension:** None (dropdown menu)
- Session Notes:** (text area)
- Buttons:** OK, Write, Cancel

UNIX Host



The **Save Snapshot** dialog box contains the following panels:

Filename Panel

This read-only text box shows you the filename that will be used when you click **Write**. As you make selections in the **Output** panel, the resulting filename is changed.

Snapshot Panel

The fields in this panel are used to select which snapshot (or live buffer) to save, when to save it, and what name (**Run Title**) to associate with it.

The **Snapshot** panel controls **when** snapshots are taken, with respect to the data collection cycle. The **Snapshot** panel contains the following fields:

- **Snapshot**

Use this drop-down menu to select which snapshot to save. You can save a snapshot you have already taken, or choose to save the live buffer. The list box

includes the live buffer, any snapshots already taken, and any snapshots you have loaded from disk.

- **Write**

Use this drop-down menu to choose when data will be written after the **Write** button is clicked.

In both Windows and UNIX hosts, the following options are available:

- **Immediately, in the middle of this cycle**
This is the default setting. This setting causes the data to be written to the output file immediately after the **Write** button is clicked, regardless of the amount of data available. An error occurs if there is no data at all.
- **Every X seconds, X specified to the right**
This setting provides a way to store data periodically. Data is written every X seconds, where X is specified in the field to the right. The write field becomes available for entering the number of seconds between snapshots. The **Write** button is labeled **Stop Writing** after Data Monitor starts writing data to the files. Click the **Stop Writing** button to terminate storing data.

In **UNIX** hosts only, the following option is also available:

- **After X seconds**
This setting provides a way to store the snapshot upon a delay of X seconds after the Write button is clicked, where X is specified in the field to the right.

- **Run Title**

Use this text box to label the snapshot. The run title is stored with the data. It may be used by data-analysis programs to identify the data set.

Data Panel (Windows Hosts Only)

In a Windows host, this panel offers choices for saving displayed data, as well as causing the live buffer to be reset after writing. The options are:

- **Write signals and events**

Saves any signals and events displayed in the GUI to a specified file.

- **Write signals only**

Saves signals only to a specified file.

- **Write events only**

Saves events only to a specified file.

- **Reset buffers after writing (live buffers only)**

For live buffers only, you can check the check box to reset the buffers after taking the snapshot.

Output Panel

The **Save Snapshot** dialog box provides automatic file-naming facilities that make saving multiple data buffers convenient. The resulting pathname and filename appear in the read-only **Filename** text box at the top. The fields in this panel specify the format, path, and name of the saved snapshot file.

The options are:

- **Format**

You can save snapshot files in the following formats:

- **Data Monitor 7.10 Compressed (.ss7.gz)**

This native format, which is XML, is the default. Files written in this format can be re-loaded into Data Monitor for later viewing, for comparison to live data, or for export to other formats. Files in Data Monitor format use the extension **.ss7**. The default format is a compressed version of the Data Monitor native format, with the extension **.ss7.gz**.

- **MATLAB**

This is a commercial data-analysis program from The MathWorks, Inc. When data is saved in **MATLAB** format, two files are created—a data file with a **.mat** extension and a script file with a **.m** extension.

- **ASCII**

Human-readable ASCII format is also supported. Files written in this format have a **.txt** extension. ASCII files may be used to import Data Monitor data into many popular spreadsheet programs.



CAUTION: Files in ASCII format can become very large; be careful!

- **MATRIX_x**

This is a commercial data-analysis program from the National Instruments Corporation. When data is saved in **MATRIX_x** format, two files are created—a script file with a **.ms** extension and a data file with a **.xmd** extension.

- **Directory**

Click the browse ("...") icon to select the path for the output file.

- **Base Name**

You can enter a filename in this text box. This is called the **base** filename because you can add automatic extensions, as described below.

- **Extension**

Data Monitor can automatically alter the base filename to help you identify your data later. The **Extension** drop-down menu determines how the new base names are produced.

The possible settings are:

- **None**

The **Base Name** string is not changed.

- **Time stamp**

The date and time are appended to the **Base Name**.

- **Cycle 0-9**

If the **Base Name** does not end in a digit already, Data Monitor appends a **0** to the name. After each store operation, the digit is incremented. Once the digit reaches **9**, it is reset to **0**. Using this option, the most recent ten buffers are saved with unique names.

- **Increment**

If the **Base Name** does not end in a number, Data Monitor appends a **0** to the name. After each store operation, the number is incremented, creating a unique name for each save.

Session Notes Panel

The notes you enter in this text box can help you keep track of a series of snapshots. Session notes are intended to describe the conditions over a series of collected runs or buffers. Useful session notes, for example, might be:

Session notes:

These runs employ the non-linear friction model.

Both force sensor filters are active, at 20 Hz.

Other useful notes include code fragments and information that clearly identifies the data. Any text on the screen may be pasted into the notes panel via standard cut and paste operations.

Formatting Filenames

Output filenames for snapshots consist of the word **Snapshot** concatenated with an integer number, starting at **1** and increasing by **1** with each new snapshot (see the example snapshot in the first figure above).

Loading Snapshots

Snapshots that have been saved to disk in the native Data Monitor format (**.ss7**) can be reloaded for viewing in the **Plot** or **Plot XY** windows. The **File > Load Snapshot** menu command (or the **Load** button), described in [Load Snapshot](#), p.45, displays the **Open** dialog box, where you can navigate to the snapshot file you want to load.

Exporting Snapshots in MATLAB and MATRIX_χ

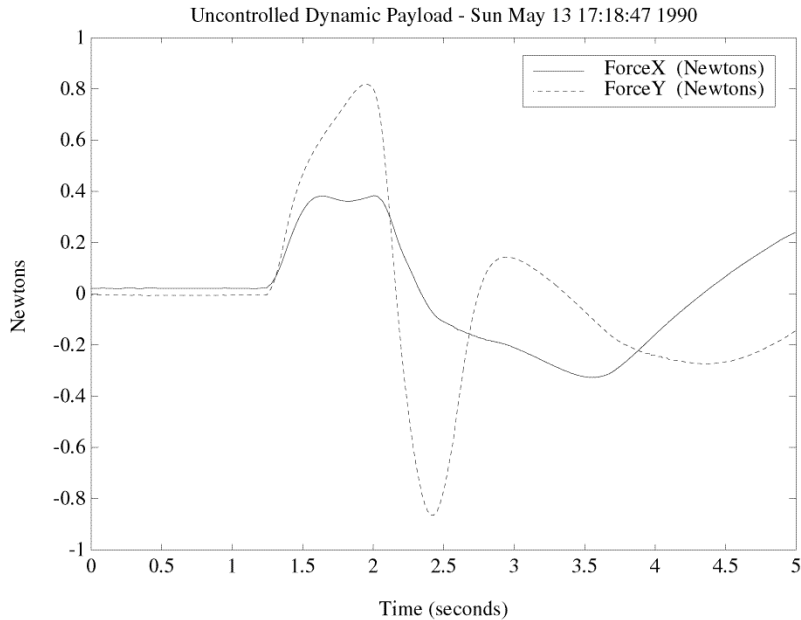
When you save a snapshot in **MATLAB** or **MATRIX_χ** format (see [Output Panel](#), p.191), two files are created:

- A script file (**.m** extension for **MATLAB**, **.ms** extension for **MATRIX_χ**)
- A data file (**.mat** extension for **MATLAB**, **.xmd** extension for **MATRIX_χ**)

Running the script file within **MATLAB** or **MATRIX_χ** loads the data and creates named vectors that correspond to each signal name saved from the buffer. This section describes the notes and variables created by the script.

MATLAB Script Example

The **MATLAB** script, **varplot.m**, provides an example of a simple **m-file** program that utilizes stored Data Monitor data (see [D. MATLAB and MATRIX_χ Examples](#)). This script creates a plot, as shown here.



Creating Variables

Running the script file generated when you save a snapshot in **MATLAB** or **MATRIX_X** format creates the following variables:

- **data**
The raw data as a two-dimensional matrix.
- **signals**
The script decomposes **data** into individual arrays that correspond to the signal in the data buffer. The variable names are the same as the original signal name, except illegal characters are replaced with underscore characters. For example, a signal named, **xdes[3]** will be converted to the variable name, **xdes_3_** in **MATLAB** or **MATRIX_X**.
- **numberOfSamples**
The length of each signal. This is the number of samples collected in a buffer cycle.

- **numberOfSignals**
The number of signals saved in the data buffer.
- **time**
A vector of time values for each sample.
- **names**
A string array of the signal names.
- **timestamp**
A vector of time values for each sample.
- **units**
A string array of the units for each signal.
- **timestamp**
A string denoting the data-collection date and time.
- **filename**
A string representing the name of the data file, without the filename extension.
- **runtitle**
The run title as displayed in the **Run Title** field.
- **notes**
A character array containing each line of the session notes.



NOTE: Limitations on the size of variable names in **MATLAB** or **MATRIX_x** may cause signal names to be truncated. If the truncation results in name clashes, then some signals may not be accessible by name as an individual array. In such cases, you still can extract the signal from the **data** matrix with a statement, such as:

```
ShoulderVel = data(:,2);
```

Creating Notes

The script file contains the following commands that create character-array variables for the run title and session notes, which are both entered on the **Save Snapshot** dialog box (see [Saving Snapshots](#), p.187):

- **runtitle**

Contains the run title.

- **notes**

Contains the session notes.

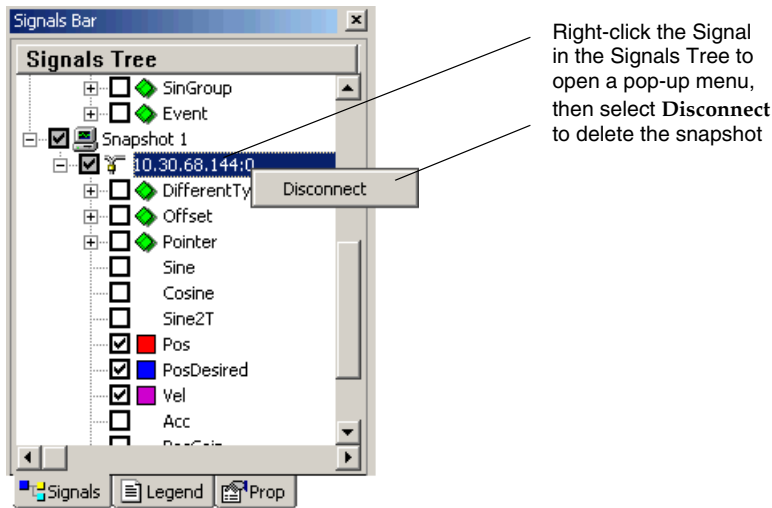
This makes it very easy to view your notes from within these analysis programs.

Deleting Snapshots

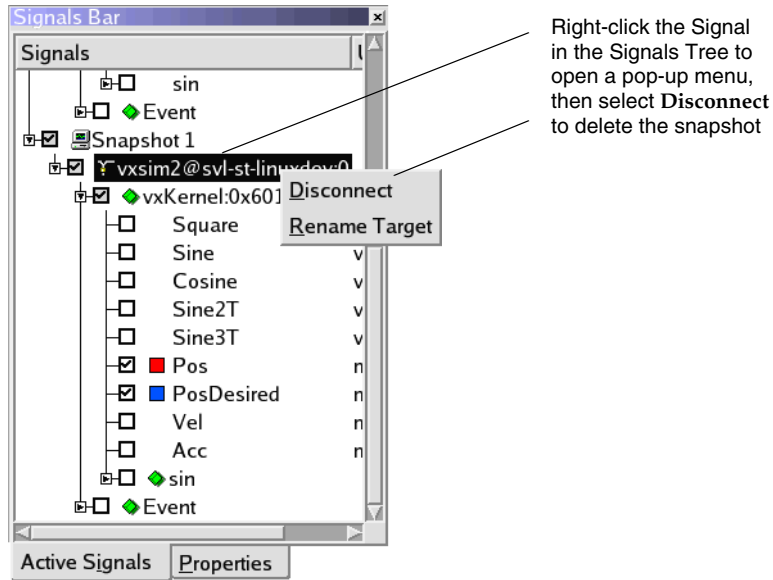
If you want to delete a snapshot from the **Plot** window, do the following:

1. Right-click the snapshot node in the **Signals Tree** to open a pop-up menu.

Windows Host



UNIX Host



2. Click **Disconnect** in the pop-up menu to delete the snapshot from the Signals Tree.
3. In a **UNIX** host only, click **Rename Target** to assign a new (perhaps more descriptive) name to the target for this snapshot.

You will not be able to retrieve this snapshot unless you previously saved it, as described in section [Saving Snapshots](#), p.187.

12

Displaying Remote Kernel Metrics

- 12.1 Introduction 199
- 12.2 Building an RKM Monitor Program 199
- 12.3 Viewing RKMs with Data Monitor 205
- 12.4 Troubleshooting 208

12.1 Introduction

Remote kernel metrics (RKMs) are operating system signals (metrics) that are dynamically collected by the **RKM_monitor** (or **rkm_monitor_linux**) target agent. The metrics can be displayed in real-time utilizing the full-color features of the Data Monitor GUI included with Workbench. For a description of signals available for monitoring, see Appendix [E. RKM Signal Definitions](#)

12.2 Building an RKM Monitor Program

To create remote kernel metrics, you must first build one or more RKM monitor programs using Workbench. The following sections describe how to do that.

On a VxWorks Target

For VxWorks, the RKM monitor supports two types of connections—**TCP/IP** and **WTX**. Typically you would use the TCP/IP connection unless you are using the VxSim simulator, in which case you should use the WTX connection. Actual VxWorks target connections can use either connection but TCP/IP is faster and does not require the debug server to be running.

Use VxWorks **Downloadable Kernel Module** (DKM) projects to manage and build modules that will exist in the kernel space. You can separately build the modules, then run and debug them on a target running VxWorks, loading, unloading, and reloading on the fly. Once your development work is complete, the modules can be statically linked into the kernel or use a file system if one is present.


Refer to the *Wind River Workbench User's Guide: VxWorks Version* for information on the available **DKM sample projects**, and using one of them to create an **RKM monitor program** whose execution can then be analyzed using Data Monitor.

Building the RKM Monitor for VxSim

The following procedure uses VxSim and the WTX connection type to demonstrate RKM monitor usage under VxWorks.

1. Start Workbench.
2. Select **File > New > Example** from the Workbench menu bar.
3. In the **New Example** dialog box that opens, select **VxWorks Downloadable Kernel Module Sample Project** and click **Next**.
4. Select **RKM (Remote Kernel Metrics) Monitor Program (WTX)** under **Available Examples** for this procedure, since VxSim is to be used as our target. Be sure to read the text in the **Information** panel to the right.
5. Click **Finish**.

A new **rkm_monitor_wtx** (Wind River VxWorks 6.6) project is created. Its elements are shown in the **Project Explorer** view.

6. Select the project, set the active build spec (using the  icon) to either **SIMNTdiab** or **SIMNTgnu** depending on your compiler, and build the project.

Downloading the Data Monitor Libraries and the RKM Monitor

The VxWorks RKM Monitor requires two Data Monitor shared libraries, **scopeutils.so** and **libscope711wtx.so**, in order to run. You must download both of these files, in the indicated order, before downloading the RKM Monitor. This can be done automatically, using the **Data Monitor Connect to Target** dialog box, or manually using the Workbench **Download** menu command.

Automatically

To load the libraries automatically, follow these steps:

1. Start a **VxSim** connection.
2. In the **Remote Systems** view, right-click the target name and select **Connect Data Monitor** from the menu that opens.
3. In the **Data Monitor Setup Options** dialog box, set the options as follows:
 - a. Set **Scope index** to the default **127** if it is not already.
 - b. Set **Connection type** to **WTX**, the type used for a VxSim target.
 - c. **Deselect** the **Start Data Monitor GUI** option at the top of the dialog box. (If you wanted to start the Data Monitor GUI now, you would leave this option selected, but in this case we are only downloading the libraries, so *be sure it is unchecked.*)
 - d. Select the **Load libraries only** option.
 - e. Click **OK**.
This automatically downloads the libraries, but the Data Monitor GUI will not be launched yet.

Manually

To load the libraries by hand, follow these steps:

1. Start a **VxSim** connection.
2. In the **Remote Systems Explorer** view, right-click the VxSim target and select **Download** in the menu that opens.
3. In the **Download** dialog box, click **Browse** and navigate to the **scopeutils.so** library, then click **OK**. Do the same for the **libscope711wtx.so** library. They are typically located in the
`install-dir/scopetools-6.0/target/arch/simmtVx64gcc4.1.2` directory.

4. In the **Remote Systems Explorer** view, right-click the VxSim target and select **Download** in the menu that opens.
5. In the Download dialog box, click **Browse** and navigate to the **rkm_monitor_wtx.out** file and click **OK**. It is typically located in the *install-dir/workspace/rkm_monitor_wtx/SIMNTgnu_DEBUG* directory.

If you expand the **SIMNT** node, the two library files described above appear in the **Remote Systems Explorer** view, and if you expand the **Kernel Tasks** node, the **RKM_monitor** task appears.

Once you have downloaded these files, you can start the monitor.

Starting the RKM Monitor Using Workbench

Once you have built the RKM monitor project, and have VxSim running (by creating a VxSim target connection), you can start the monitor as follows:

1. In the **Remote Systems Explorer** view, select the VxSim connection.
2. Select **Target > Run > Run Kernel Task** from the Workbench menu bar.
3. In the Main tab view, select the VxSim target in the **Connection to use** drop-down menu, and enter **RKM_monitor** in the **Entry Point** field.
4. Click **Run**.
5. Expand **Kernel Tasks** under **SIMNT** in the **Remote Systems Explorer** view and you should see the **RKM_monitor** kernel task running. You can also enter the command **RKM_list** in the VxSim window for additional verification.

Controlling the RKM Monitor Using VxSim

In the VxSim window, you can use the following commands to control the RKM monitor:

- **RKM_monitor**—Starts an RKM monitor with the default options.
- **RKM_list**—Lists all monitors which are currently running.
- **RKM_shutdown**—Stops all RKM monitors.
- **RKM_stop index**—Stops the RKM monitor using index number **index**, where index is a number from 0 to 127 (the default).
- **RKM_monitor options**—Starts an RKM monitor with options, specified within double-quotes.
- **RKM_monitor -help**—Displays the RKM monitor options.

For example, the following sequence of commands starts an RKM monitor with with a non-default port, shows that it is running, and then stops it.

```
-> RKM_list
value = 0 = 0x0
-> RKM_monitor "-index=125"
value = 0 = 0x0
-> RKM_list
Monitor[125] is running
value = 0 = 0x0
-> RKM_stop 125
value = 0 = 0x0
-> RKM_list
value = 0 = 0x0
->
```



NOTE: The RKM_monitor in the above example was started with a specific index (port) number **125** (the default is **127**). Using this feature, you can run multiple monitors simultaneously on your target, which has several advantages. You might want to configure one monitor, for example, to collect a few signals for all processes at a low sampling frequency, and configure another to sample a complete set of metrics for a few processes at a high frequency.

12

By specifically selecting the signals you want to monitor, you can reduce the memory, CPU, and network resources required to monitor the large set of signals selected by default. In addition, the source for the RKM monitor is included so you can create versions that monitor specific signals that are not made available by the default configuration, or even monitor specific portions of your application.

For example, to start an RKM monitor with index 125 to monitor only the system metric tracking the number of tasks on the system, taking 10 samples every second:

```
->RKM_monitor "-index=125 -sysmetrics tasks -samples=10"
```

When you attach Data Monitor to the RKM monitor invoked for specific metrics, you will be able to view only those metrics in Data Monitor.

On a Linux Target

The RKM monitor acquires its data from the **/proc** filesystem on the target. If you do not have a **/proc** filesystem on your target, you may simply need to mount it, or you may need to rebuild your kernel to include it.

To mount the **/proc** filesystem, use the **mount** command as follows:

```
# mount -t proc proc /proc
```

If your kernel does not have **/proc** support built-in, you must re-build the kernel and enable it in the **File system** configuration section of your kernel configuration tool.

The RKM monitor agent is supplied in a Workbench sample project. Use the following procedure to build the RKM monitor and then run it on your Linux target.



NOTE: This procedure assumes the results of your project build are available on the target, for example by locating your workspace on a shared NFS mount. It also assumes you have created a connection to the target in the Workbench **Remote Systems Explorer** view and have specified the target-host root filesystem mapping.

1. Start Workbench.
2. Select **File > New > Example > Native Sample Project**, and click **Next**.
3. Select **The RKM (Remote Kernel Metrics) Monitor Program**, and click **Finish**.



NOTE: Be sure you select the **C++-Linker** if you are building the **rkm_monitor** example project.

4. In the **Project Explorer** view, right-click **rkm_monitor_linux** and select **Build Project**.
5. Select **Run > Run**, and select **Process on Target**.
6. In the **Main** tab, name it something like **rkm_monitor_linux** and choose your target connection from the **Connection to use** pull-down menu.
7. Click **Run**, and **rkm_monitor** starts on the target as shown in your **Debug** view.

Running the RKM Monitor From the Command Line

You can also start the RKM monitor by specifying it on the command line. To see the various options available, enter the following from the directory containing the **rkm_monitor_linux** executable that you built:

```
$ ./rkm_monitor_linux -help
```

For example, to start the RKM monitor with a different index value, say 125 instead of the default 127, enter:


```
$ ./rkm_monitor_linux -index=125 &
```

In this way you can run multiple monitors which has several advantages. You might want to configure one monitor, for example, to collect a few signals for all processes at a low sampling frequency, and configure another to sample a complete set of metrics for a few processes at a high frequency.

By specifically selecting the signals you want to monitor, you can reduce the memory, CPU, and network resources required to monitor the large set of signals selected by default. In addition, the source for `rkm_monitor_linux` is included so you can create versions that monitor specific signals that are not made available by the default configuration, or even monitor specific portions of an application.

As another example, you might only want to monitor memory usage for the root user, taking 10 samples every second:

```
$ ./rkm_monitor_linux -samples=10 -processes user=root -sysmetrics memory &
```

When you attach Data Monitor to the RKM monitor invoked as shown, you will only be able to view root memory usage.

12

12.3 Viewing RKMs with Data Monitor

Each RKM monitor program created using Workbench, as described in Section [12.2](#) above, can be used to generate any or all of the metrics signals described in Appendix [E. RKM Signal Definitions](#). RKM monitor programs are created to display their output specifically using the Data Monitor real-time graphical monitoring tool. In the case of multiple RKM monitor programs you may wish to run concurrently, each must be connected to Data Monitor using a unique port connection (in the range of 0-127).

For a list of the metrics available for viewing, use the `RKM_monitor` help by typing:

```
->RKM_Monitor "-help"
```

You can also refer to Appendix [E. RKM Signal Definitions](#) for a description of all the RKM metrics.

On a VxWorks Target

Starting Data Monitor GUI with a WTX Connection

You can start the Data Monitor GUI using the Workbench main menu (see [Starting Automatically](#), p.11). Be sure to select **Load libraries only** in the **Connect to Target** dialog box, and also remember to check that **Start Data Monitor GUI** is also now selected.

Alternatively, you can create a desktop icon to start the Data Monitor GUI and connect it to the target. Because the WTX version requires some VxWorks environment variables to be configured, it is easier to create an icon which invokes the Data Monitor program using a VxWorks Development shell.

For example:

```
C:\PATH\wrenv.exe -p vxworks-6.6 scope.exe -index 127 -verbosity 0  
-tgtsvr vxsim0@HOST -wtxMode
```

Note that you have to enter the correct information for your path to wrenv.exe and also your host information.

Using Data Monitor to View Remote Kernel Metrics

The primary navigation tool for Data Monitor is the **Signals Tree** in the upper-left corner of the Data Monitor GUI. When RKM signals are installed and selected, they appear in the **Signals Tree** (see [Working With Signal Trees](#), p.82). As an example for viewing these signals with Data Monitor, do the following:

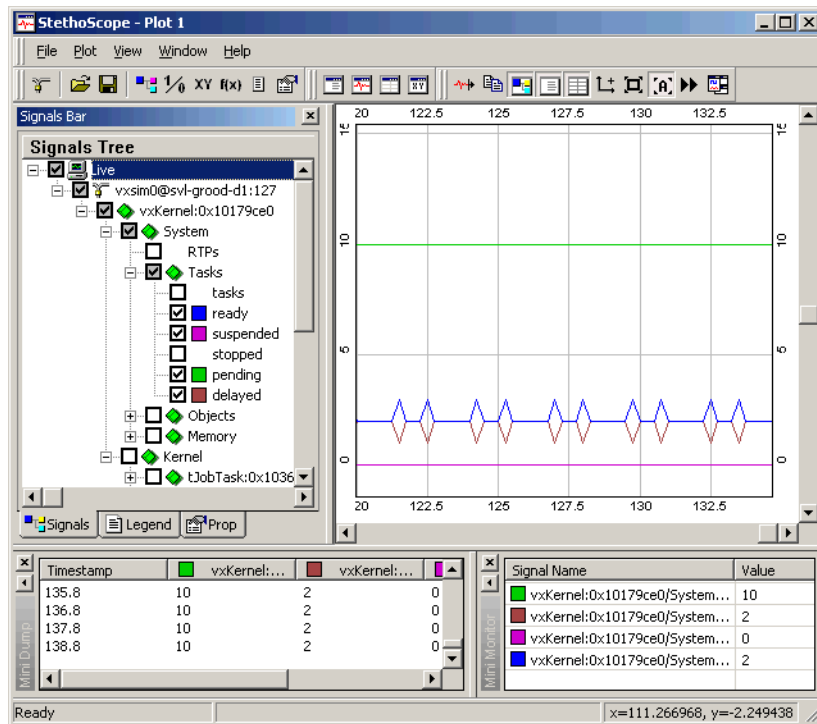
1. In the Signals Tree, expand **vxsim**, **vxKernel**, **System**, and then **Tasks**.
2. Check the **Tasks** group to automatically check all of the metrics under **Tasks** (that is, **tasks**, **ready**, **suspended**, **stopped**, **pending**, and **delayed**), or select only the desired signals within the **Tasks** subgroup.

These signals show in the GUI as the total number of tasks running, followed by the number of tasks currently in each of the remaining five states, all displayed dynamically with respect to time in the real-time graph.

3. Select the **Zoom to Fit** icon to adjust the graph window boundaries to include all the signals.

The signals selected in the Signals Tree should now appear in the same graph, being updated in real-time.

The example RKM_monitor data described above, and displayed in Data Monitor, is shown here.



Note that a color swatch next to each signal in the Signals Tree has a unique color associated with that signal. This color is used for the corresponding line in the graph. The MiniMonitor view lists the current values of monitored signals and the MiniDump view lists the value of each signal at each sampling.

By default, when any new processes are started they are added to the monitor. Note, however, that the buffer is reset when new signals are added, so you lose the history of what you had been monitoring. You can avoid this by, for example, monitoring only your own processes with appropriate command line options.

On a Linux Target

Attaching Data Monitor to the RKM Monitor

For a Linux target, use the following procedure to view the remote kernel metrics in the Data Monitor GUI.

1. With **rkm_monitor** running on the target, start Data Monitor (refer to the steps outlined in [2.3 Starting Data Monitor](#), p.10 if needed).
2. Select the correct target connection from the pull-down menu in the Linux **Data Monitor Setup Options** dialog box.
3. Enter **127** as the index value. This is the default to use for Data Monitor with the RKM monitor. Use a different index number if you want to run another monitor on the target; this allows you to run multiple monitors on the target, selecting them by index number.
4. Click **OK**.

The Data Monitor GUI opens.

Using Data Monitor to View Remote Kernel Metrics

The Data Monitor GUI displays data nearly identically the same for a Linux target as for a VxWorks target. Therefore, the display of RKM_monitor data described in [On a VxWorks Target](#), p.206 above accurately and adequately serves to describe the results on a Linux target.

12.4 Troubleshooting

This section describes specific error conditions and their remedies, some for VxWorks targets only, and some applicable to all targets.

On a VxWorks Target

Loading Libraries Only

If you have difficulties displaying RKM data from a VxWorks target, the first step in debugging is to check if you started Data Monitor with the **Spawn sampler task** option selected in the **Data Monitor Setup Options** dialog box (see [Starting Automatically](#), p.11). The RKM module starts up with its own self-contained sampler task, so if you started Data Monitor with this option, failure of the RKM module is certain. The error message:

```
Signal Buffer Signal New: Out of memory. Increase signal buffer size.
```

will help verify this as the problem. The **Spawn sampler task** option creates only a small (2048-byte) signal buffer, enough for only a handful of signals. When Data Monitor is started first, then handed off to the RKM monitor afterwards, this buffer size cannot be increased during execution.

If this is the source of the trouble, restart Data Monitor with the **Load libraries only** option selected. The RKM monitor can then specify the signal buffer size that it needs.



CAUTION: When viewing RKM signals, it is strongly recommended that the **Load libraries only** option always be selected in the **Data Monitor Setup Options** dialog box when you start the GUI in order to avoid encountering this error condition.

On All Targets

Failed to install signal

12

The following error message may be observed in the Data Monitor GUI while installing RKM signals on a VxWorks or Linux target:

```
Error: Failed to install signal "x".
      There are currently Y installed signals (maximum of Z)
      Approximate signal buffer usage: M out of N.
```

where **x** is the name of the signal it was unable to install, **Y** is the number of signals successfully installed, and **Z** is the maximum number of signals the monitor can handle without being restarted with a larger buffer size. RKM tells you how many bytes (**M**) are being used out of a signal buffer size of **N** bytes.

If you receive the above mentioned error message, the solution is to call `RKM_stop`, then restart RKM monitor using `(-signalbuf=xxxx)` where `xxxx` is a buffer size (in bytes) sufficient to handle the number of signals you expect to view.

Unpredictable Results with Multiple Targets

To help prevent multiple connected target data from outpacing each other on the plot screen when they are sampling at the same rate, Data Monitor attempts to automatically sync the sweep rates of all the targets. In many cases this will be a straightforward operation, and the resulting plot will have lines reaching the right-hand end of the screen at the same time, ready to erase the screen and start the next 20 seconds of plotting.

Sometimes, however, one or more targets will be sampling at a rate *different* from what they say they are sampling at (52Hz when they said 50Hz, for example). Using the samples from the first target to reach the end of the screen, Data Monitor autosyncs by calculating the time difference between that and the last sample received from each of the other targets. Each of the other targets is then shifted forward in time to match, on a per-plot basis. This helps because, since samples are sometimes not received at exactly the rate specified, they may be plotted ahead of, or behind, where they would have if they were timestamped.

If you do not wish to see the data synchronized this way, plot the different targets in different windows. That way, you will always be able to see the samples plotted without any shifting.



NOTE: The above explanation holds true only for non-**Strip Chart** mode plotting.

In **Strip Chart** mode (see [Strip Chart](#), p.122), Data Monitor does not autosync the sampled data, as there is no end to the plotting screen (it is a continuous scrolling area). Therefore, when targets are not providing samples at exactly the expected rate, the samples from one or more targets will eventually become more than a screen behind, and you will no longer see them. The location where a sample appears at on the screen is a simple **sample #/sample rate** calculation, and thus everything depends upon getting the right number of samples. In this mode the buffers are initially cleared, so the sample at time **0** will be at the same time for all targets.

For continuous saving of data, Data Monitor simply writes the samples in hand when a **cycle end** event is declared. This is usually driven from one of the targets.

13

Using a VxWorks Target

13.1 Introduction	211
13.2 ScopeProbe Requirements	212
13.3 VxWorks Targets	213
13.4 Troubleshooting	220

13.1 Introduction

This chapter contains specific information applicable to the building and using of target programs running on a VxWorks OS.

As an alternative to signal installation (see [15. Installing Signals](#)), target programs may instead be instrumented with commands from the Wind River Data Monitor API, then recompiled. For details on the Data Monitor VxWorks API, see [A. API Reference: VxWorks](#).

13.2 ScopeProbe Requirements

In general, to build an application that is instrumented with the ScopeProbe API, you need to:

Add the following include file to your code:

```
#include "scope/scope.h"
```

Add the following **DEFINE** to your makefile or project:

```
-D RTI_VXWORKS
```

Add include paths so the compiler can locate the **scope.h** include file:

```
-I $WIND_SCOPETOOLS_BASE/target/include/share/scope
```

Link or load the following libraries:

```
libutilsipz.a  
libxmlparsez.a  
libscope711tcpz.a  
  
libutilsipz.lib  
libxmlparsez.lib  
libscope711tcpz.lib
```

For TCP/IP:

```
scopeutils.so (contains libutilsip.so and libxmlparse.so)  
libscope711tcp.so
```

For WTX:

```
scopeutils.so (contains libutilsip.so and libxmlparse.so)  
libscope711wtx.so
```



NOTE: A detailed implementation of some of these steps, as applied to the VxWorks demonstration code **vxdemo.c** and its **makefile**, is shown in [C. Data Monitor Demo Program](#).

13.3 VxWorks Targets

The following sections discuss specific requirements for a VxWorks target platform.

Building

The Wind River Run-Time Analysis Tools installation installs the required header files into the Run-Time Analysis Tools installation tree. You do not need to link the ScopeProbe libraries directly to your application. Instead, the ScopeProbe libraries are loaded automatically when you click the Data Monitor button on the Workbench toolbar.

The folder `WIND_SCOPETOOLS_BASE\target\src\vxworks\scopedemo`, contains the source code for the demo program **vxdemo.c** that you can start from Workbench, where `WIND_SCOPETOOLS_BASE` (an environment variable of the same name) is the root of the tree where you installed the Run-Time Analysis Tools. The directory also contains a **makefile** that you can use to compile the code. We suggest you use this makefile as a template for compiling your code instrumented with ScopeProbe API.



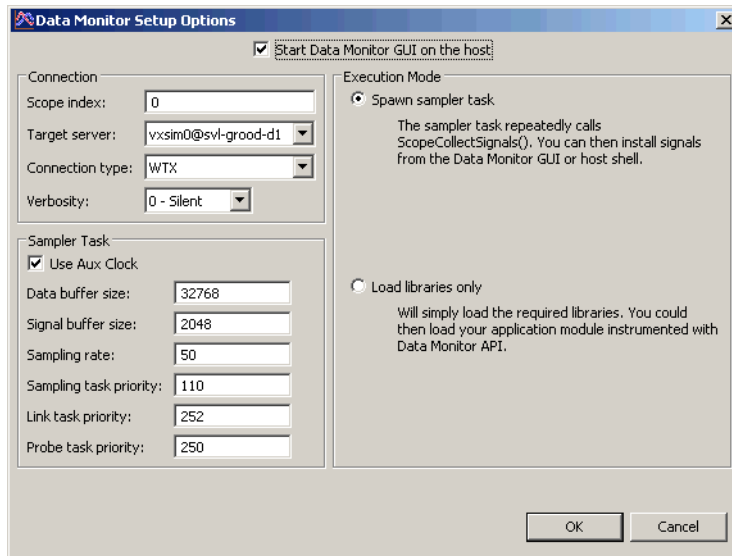
NOTE: For instructions on how to build **vxdemo.c**, see [C. Data Monitor Demo Program](#).

Automatic Loading and Running

Clicking the Data Monitor button on the Workbench toolbar starts the Data Monitor GUI automatically, with no manual intervention. To start Data Monitor automatically, follow these steps

1. In Workbench, click the **Data Monitor** button, then click **OK**.

This opens the **Data Monitor Setup Options** dialog box, allowing you to configure the Data Monitor and ScopeProbe initialization parameters.



2. In the **Data Monitor Setup Options** dialog box, specify whether you want to run an asynchronous sampler task, or just load the libraries, by specifying the following parameters:

Spawn sampler task

Spawns a task on the target that collects data for any signals that you install (either from the Data Monitor GUI on the host, or from the host shell) using the specified **Scope index** value. This task repeatedly calls **ScopeCollectSignals()**, based on its own timing (you can also use the auxiliary clock), so it is asynchronous with the running of your application. This option requires you to provide additional initialization parameters, such as **Sample Buffer Size**, **Signal Buffer Size**, **Sampling Rate**,

or,

Load libraries only

Deselecting **Start Data Monitor GUI on the host** check box, and clicking Load libraries only, loads the required applications libraries without having to restart the Data Monitor GUI. You can then load your application module instrumented with the ScopeProbe API.

For all modes, specify these settings in the **Connection** panel:

- a. **Scope Index**
Distinguishes the different instances of ScopeProbe daemons running on the same target. Up to 128 different instances may be started on a target, so the index can range from **0** to **127**. The index specified here is used to initialize the Data Monitor GUI and, if selected, the demo program or sampler task.
- b. **Target server**
Select a target server from the drop-down list of discovered target servers. If the list is empty, you do not have a target server running and you must create one first. For details on how to configure and start a target server, see the *Wind River Workbench User's Guide*.



NOTE: The Data Monitor setup script uses the currently selected target-server name, **target@tgtsvrHost**, to determine the host name of the target.

- c. **Connection type**
Use the default (**TCP/IP**), or choose **WTX** protocol only if your target does not have TCP/IP support.
 - d. **Verbosity**
Controls the volume of status and information messages written to the log file. **Verbosity** has the following options to choose from:
 - 0** (silent) - Displays only warning and error messages (most restrictive)
 - 1** - Displays warning, error, and workflow messages.
 - 2** - Displays warning, error, and greater volume of workflow messages.
 - 3** (verbose) - Displays all system messages (most verbose)
3. If you select **Spawn Sampler Task**, accept the defaults or specify values for these parameters in the **Sampler Task** panel:
 - a. **Use Aux Clock**
This check box causes the sampler task to attach a semaphore to the VxWorks auxiliary clock for periodic timing. You must clear this box if you do not have an auxiliary clock, or if you have another application using the auxiliary clock. If this option is not checked, the sampler task calls **taskDelay()** to obtain pseudo-periodic timing.

- b. **Data buffer size**
Specifies the size (in bytes) of the buffer to be allocated on the target for collecting data samples. For a description of this buffer, see [Target Buffers](#), p.249. The default is **32768** bytes.
 - c. **Signal buffer size**
Specifies the size (in bytes) of the buffer to be allocated on the target to store signal information. For a description of this buffer, see [Target Buffers](#), p.249. The default is **2048** bytes.
 - d. **Sampling rate**
Specifies the rate (number of times per second) at which the sampler task calls **ScopeCollectSignals()** to collect data for the signals installed to the specified **Scope Index**. The default is **50** Hz.
 - e. **Sampling task priority**
Sets the task priority for the sampler task. The highest priority is **0**, **255** is the lowest. The default is **110**.
 - f. **Link task priority**
Sets the task priority for the link task. The default is **252**.
 - g. **Probe task priority**
Sets the task priority for the probe task. The default is **250**.
4. Click **OK** to load the appropriate libraries onto the target and initializes the target. If the **Start Data Monitor GUI on the host** option is checked, the Data Monitor GUI will appear shortly, with an open **Plot** window.

After selecting parameters and clicking **OK** in this dialog box, the following actions are performed:

- The required VxWorks libraries are loaded onto the target.
- The libraries are loaded onto the target.
- ScopeProbe is initialized on the target.
- The Data Monitor GUI is started on the host.
- Optionally, the demo program can be loaded and started on the target.

Verifying Target Initialization

To verify that the target has been initialized:

1. From the Workbench toolbar, right-click your target, select **Target Tools > Host Shell** to bring up a shell.

2. Type the command "i" in the shell to print a list of running tasks.
3. If you are not using WTX mode, the following tasks should be listed:
 - **tProbeDaemon**
 - **tLinkDaemon**
 - **tSamplerTask** (if you chose to run the sampler task)

Verifying Target Connection

To verify that the Data Monitor GUI is connected to your target:

1. Verify that the target appears in the **Signals Tree** of the **Plot** window.
To open a **Signals Bar** panel if one is not already displayed, see [Signals Bar](#), p.116.
2. Select the signals you want to display in the **Plot** window (see [4. Using the Signal Manager](#)). If you are not running the demo program, make sure you have installed signals already (see [4. Using the Signal Manager](#), and [Registering and Activating Signals](#), p.249).
3. Verify that the selected signals are plotted in the **Plot** window.

13

Manual Loading and Running

If you do not use Workbench, or if automatic loading fails, you will have to determine which libraries to load yourself, load them, then initialize Data Monitor manually. Manual loading of Data Monitor is more involved than automatic loading. We strongly recommend you create a target-shell script that you source from the VxWorks shell to accomplish Data Monitor loading and initialization.

To load a library, type in the host 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 the Run-Time Analysis Tools, *targetArch* reflects the processor and VxWorks version you are using, and *library* is the library to load. The *targetArch* string format is:

```
cpu os osversion compiler compilerversion
```

An example would be:

```
ppc85xxGPP1.2gcc4.1.2
```

Refer to the release notes included with your product suite for a list of supported architectures.

The "1" (number one) 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 Run-Time Analysis Tools with your code.

[Table 13-1](#) lists the libraries needed by the ScopeProbe daemons.



CAUTION: The library files shown in this table must be loaded in the order shown.

Table 13-1 **Target Libraries**

Target Configuration	Target Libraries
VxWorks 6.6	scopeutils.so (contains libutilsip.so, libxmlparse.so) libscope711tcp.so or libscope711wtx.so vxdemo.so (for demo only)

The following sections describe in detail how to determine which libraries to load.

Loading the Run-Time Analysis Tools Utilities Library

Data Monitor requires the **ScopeUtils** library.

Load **scopeutils.so** using:

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

Loading the TCP/IP Library

If your target supports TCP/IP, load **libscope711tcp.so** using:

```
ld 1 < WIND_SCOPETOOLS_BASE/target/arch/targetArch/libscope711tcp.so
```

If your target does not have TCP/IP enabled, load **libscope711wtx.so** using:

```
ld 1 < WIND_SCOPETOOLS_BASE/target/arch/targetArch/libscope711wtx.so
```

Loading the Demo Library

If you wish to load and run the demo, load it using:

```
ld 1 < WIND_SCOPETOOLS_BASE/target/arch/targetArch/vxdemo.so
```

To initialize the demo, type the following function call at the VxWorks shell or place it into a target-shell script:

ScopeDemo(*useAuxClk*, *scopeIndex*, *verbosity*)

where the parameters have the following meanings:

useAuxClk

Set to **0** to use **taskDelay()** for timing, or **1** to use the VxWorks auxiliary clock.

scopeIndex

Specifies the scope index value.

verbosity

Specifies the volume of debug messages printed by the ScopeProbe daemons. These messages appear in the shell from which you run **ScopeDemo()**. A value of **0** causes only error messages to be printed. Increasing this to a value from **1-3** increases the amount of warning and debug messages.

These parameters are described in detail in [d.Verbosity](#), p.215.

Starting the Sampler Task

If you wish to start the sampler task (rather than run the demo) for asynchronous sampling of signals, type the following function call at the VxWorks shell or place it into a target-shell script:

ScopeSamplerTaskCreate(*dontUseAuxClk*, *scopeIndex*, *sampleBufferSize*, *signalBufferSize*, *samplingRate*, *verbosity*)

where the parameters have the following meanings:

dontUseAuxClk

Set to **1** to use **taskDelay()** for timing, or **0** to use the VxWorks auxiliary clock.

scopeIndex

Specifies the scope index value.

sampleBufferSize

Specifies the size (in bytes) of the buffer to be allocated on the target for collecting data samples. For a description of this buffer, see [Target Buffers](#), p.249.

signalBufferSize

Specifies the size (in bytes) of the buffer to be allocated on the target to store signal information. For a description of this buffer, see [Target Buffers](#), p.249.

samplingRate

Specifies the rate (number of times per second) at which the sampler task calls **ScopeCollectSignals()** to collect data for the signals installed to the specified *scopeIndex*.

verbosity

Specifies the amount of debug messages printed by the ScopeProbe daemons. These messages appear in the shell from which you run **ScopeDemo()**. A value of 0 specifies that only error messages are printed. Increasing the value increases the amount of messages. See [d.Verbosity](#), p.215 for specific information on selecting verbosity values.

Example Target Script

The following is a complete example of a target-shell script to load and initialize Data Monitor and the demo on a VxWorks target. The target in this example supports TCP/IP.

The example script for Workbench 2.x / VxWorks 6.6:

```
ld 1 < WindRiver/scopetools-6.0/target/arch/ppcVx6.3gcc4.1.2/scopeutils.so
ld 1 < WindRiver/scopetools-
6.0/target/arch/ppcVx6.3gcc4.1.2/libscope711tcp.so
ld 1 < WindRiver/scopetools-6.0/target/arch/ppcVx6.3gcc4.1.2/vxdemo.so
sp ScopeDemo
```

Starting the Data Monitor GUI Manually

To start the Data Monitor GUI manually, see the instructions in [Starting Manually](#), p.14, or consult the reference documentation (paper manual or online manual) for the topic **scope**.

13.4 Troubleshooting

Load Errors

If you have trouble loading the object files onto your VxWorks target, such as the error message:

```
API_FILE_NOT_FOUND,
```


check the following:

- You are able to **ping** the target over the network.
- If you are using NFS, check that the file system is mounted (use **nfsDevShow**).
- Your target has permission to read the object files from the file server.

If none of these suggestions resolve the problem, it may be that your target system is slow, or has intermittent response. Try loading the Run-Time Analysis Tools modules manually using a shell window and the **"ld"** command (see [Manual Loading and Running](#), p.217).

Connection Failure

If the **Data Monitor Plot** window status message is, **Target target not responding**:

- Make sure your network is configured properly. You must be able to establish a network connection to your target via **ping** or **rlogin** before Data Monitor will work.
- On VxWorks targets, if these tests are successful, type **i** on the target processor and make sure the **tProbeDaemon** and **tLinkDaemon** tasks are active.
- Make sure you are using the same scope index on the target and host.

No Response from Target

If the **Data Monitor Plot** window status message displays, **Target target not responding**, or Data Monitor exits with the message:

```
scope: target is connecting but not responding.
```

then one of the following has occurred:

- Another Data Monitor GUI is already connected to your target using the same index
- The **tLinkDaemon** task is being starved for processing time.
- The network is not configured correctly.
- The target is loading an incompatible version of ScopeProbe.

Multiple Connections

When you are connecting to multiple targets, you must ensure that each target is connected by a unique index (from 0 to 127). If the target you to which you are trying to connect has a host GUI already connected to that index (either yours or a different user), this creates the error condition. You must use a different index number.

The use of an index number in making a target connection is discussed in several places in [2.3 Starting Data Monitor](#), p. 10. The assignment of index values in your target code is explained in [Scope Index](#), p. 249. The **Legend** tab view shows each connected target name and index, as shown in [Signals Bar](#), p. 26.

Starvation

In VxWorks, the second condition can be tested by executing this command on the target:

```
taskSpawn("test",255,0x1c,12000,printf,"Not starved.\n")
```

This tries to spawn a low-priority task that only prints a message. If the message, **Not starved**, does not print, then starvation is the problem. Starvation can be cured by incrementing the priorities of **tLinkDaemon** and **tProbeDaemon** tasks (via **taskPrioritySet()**) or ensuring that higher-priority processes do not use all the available CPU. This is rarely a problem for VxWorks targets with a Windows host.

Network Configuration

A **ping** or **rlogin** test suffices to test proper network connection.

Version Mismatch

The ScopeProbe version can be printed via **ScopePrintVersion()**. The version of the Data Monitor graphical interface is displayed on the **Help > About** dialog box.

None of the Above

Finally, try increasing the verbosity value. Call **ScopeInitServer()** (or start the demo or sampler task) with verbosity set to 1; start the Data Monitor GUI with the **-v 1** flag. The output messages may help you pinpoint the problem.

No Data

If the connection appears to be normal, and the **Signals Tree** in the **Signals Bar** on the **Plot** and **Dump** windows show installed signals but no data appears, check the following:

- **ScopeCollectSignals()** is being called to sample data.
- **Triggering** is set up correctly and that trigger conditions are occurring.
- The **tLinkDaemon** task is not being starved for processing time.
- You are viewing an active buffer.

Sampling

Under VxWorks, test for this condition by placing a breakpoint at **ScopeCollectSignals()** from the VxWorks shell.

Triggering

Disable triggering from the **Triggering** dialog box.

Starvation

Starvation can be tested as described in *No Response from Target*, p.221. Raising the priorities of **tLinkDaemon** and **tProbeDaemon** tasks or insuring that higher-priority processes do not use all the available CPU will alleviate the problem. Again, this is rarely a problem for VxWorks targets with a Windows host.

None of the Above

Finally, make sure the window you are using is displaying the live buffer and not a stored static buffer (snapshot). Also make sure the range is wide enough to plot data on the screen (try the **Zoom to Fit** button, or from the **View** menu).

14

Using a Linux Target

[14.1 Introduction](#) 225

[14.2 Building Your Application](#) 225

14.1 Introduction

This chapter contains specific information applicable to the building and using of target programs running on a Linux OS.

As an alternative to signal installation (see [15. Installing Signals](#)), target programs may instead be instrumented with commands from the Wind River Data Monitor API, then recompiled. For details on the Data Monitor API, see [B. API Reference: Linux](#).

14.2 Building Your Application

In general, the steps to build an application that is instrumented with the Data Monitor API requires you to do the following:

1. Add an include file to your target code.

2. Instrument your target code
3. Link libraries during compilation.
4. Compile your target code.
5. Test your finished application.

The following sub-sections examine each of these steps in greater detail.

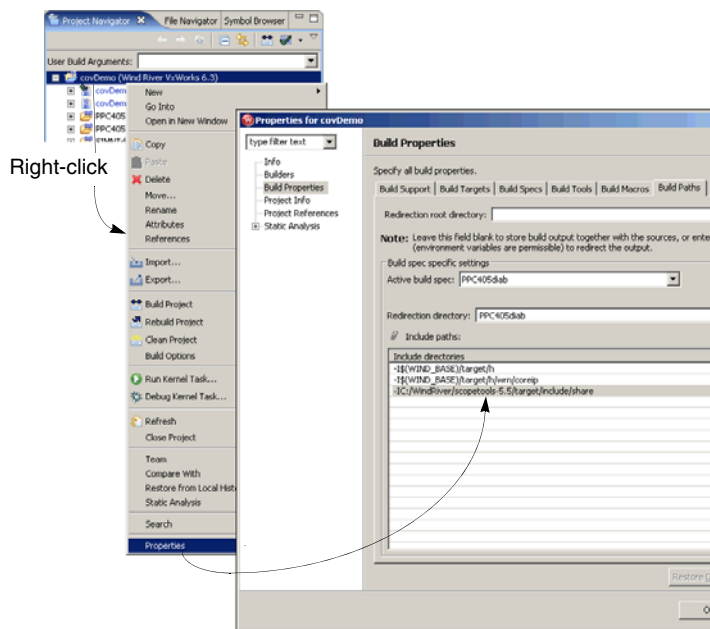
Adding Include Files

Add include paths to your compilation lines so the compiler can locate the **scope/scope.h** include file, as well as the necessary library files:

`-I WIND_SCOPETOOLS_BASE/target/include/share/`

where `WIND_SCOPETOOLS_BASE` (an environment variable of the same name) is the root of the tree where you installed the Run-Time Analysis Tools.

This path is added at the **build properties->build paths** tab view of the project **Properties** dialog box, in the **Include directories** table.



Instrumenting Target Code

Instrument your target code by adding the following commands in the code:

```
#include <scope/scope.h>

Call ScopeInitServer once at startup

Call ScopeInstallSignal once for each variable to monitor

Call ScopeCollectSignals to gather data

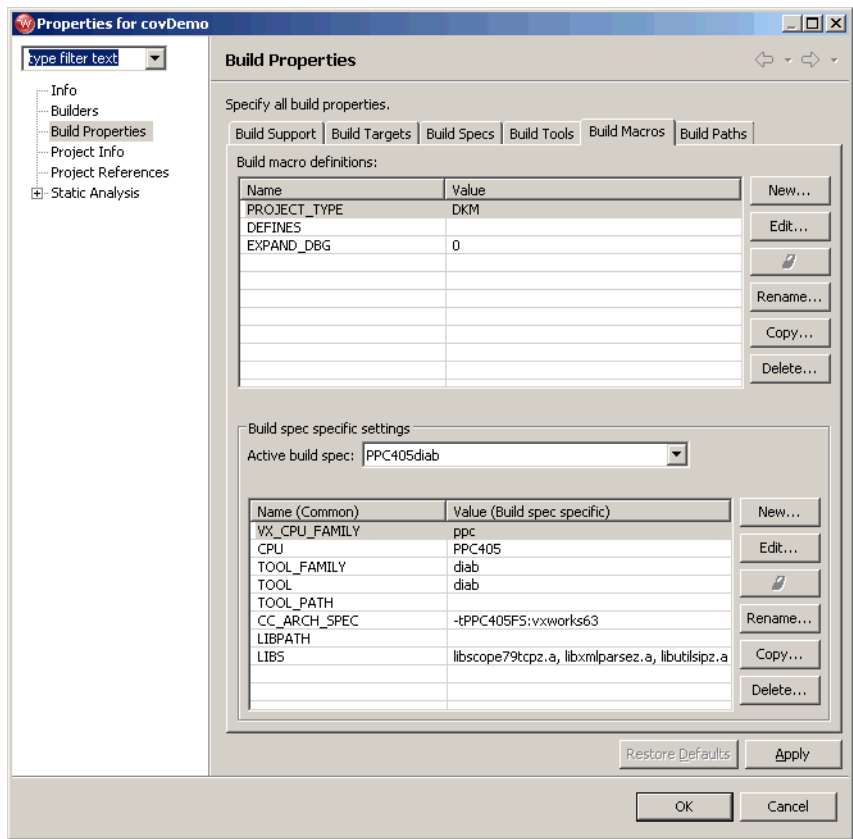
Call ScopeChangeSampleRate to match the frequency of the calls to
ScopeCollectSignals()
```

Adding Libraries

Include the required Run-Time Analysis Tools libraries by adding the following libraries to your link line (in the order shown):

- **libscope711tcpz.a**
- **libxmlparsez.a**
- **libutilsipz.a**

These directories are added at the **build properties->build macros** tab view of the project **Properties** dialog box, in the **LIBS** name field.



There needs to be a space between each library name, and each name must be entered with a complete pathname.

For example, in a **Windows** host, enter:

```
$INSTALLDIR/target/arch/ppc85xxGPP1.2gcc4.1.2/libscope711tcpz.a
```

In a **UNIX** host, enter:

```
$WindRiver/scopetools-  
6.0/target/arch/ppc85xxGPP1.2gcc4.1.2/libscope711tcpz.a
```

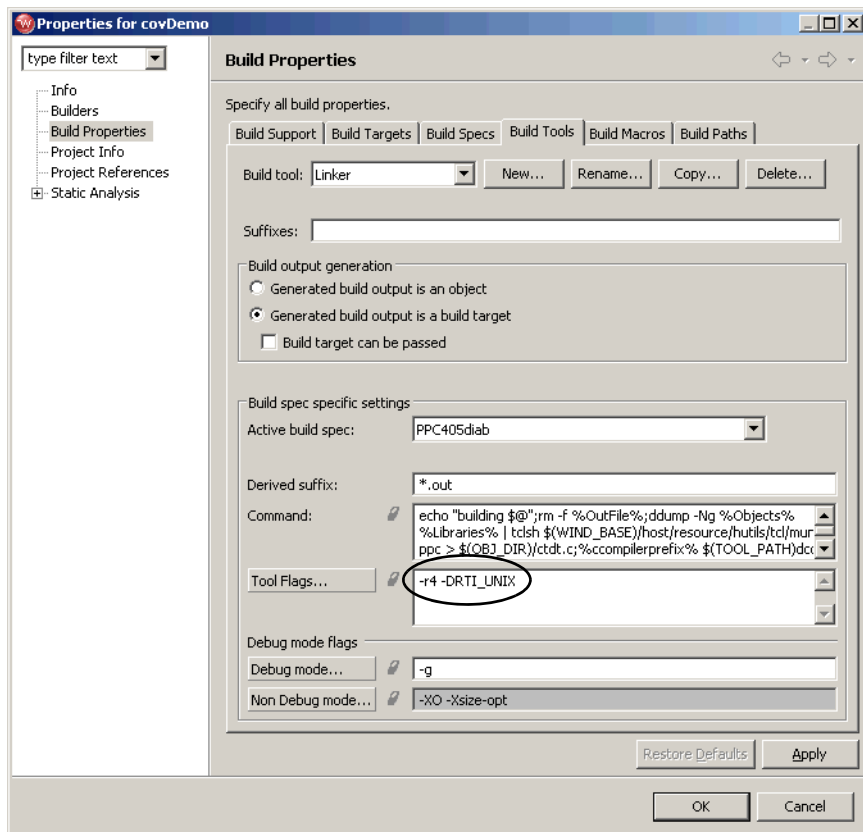

Compiling Target Code

This step is normally required to instrument your code directly with the Data Monitor API routines. You can, however, use the process sampler program described in [15.4 Code Instrumentation Alternative](#), p.241 as an alternative to instrumenting your own code.

Depending on the compiler you use to compile your code, you must be sure that the following **define** is added as a switch to the compiler directive in order to compile with Run-Time Analysis Tools headers:

-DRTI_LINUX

Enter this define switch in the **build properties- > build tools** tab view, in the **Tool Flags** entry field (with the **C/C++Compiler** Build tool selected), as shown (circled) here.



Testing Your Application

Run the resulting Data Monitor demonstration program on your target, testing the instrumentation using the Data Monitor GUI. Consult [2. *Getting Started*](#), if you are not yet comfortable with starting Data Monitor and connecting to targets.

You can run the demo program to connect with a Data Monitor GUI to see how Data Monitor works. You can find source code for the demo program, **scopedemo.c**, in the host directory:

`WIND_SCOPETOOLS_BASE/target/src/linux/scopedemo_linux.`

This directory also contains a **makefile** template that you can use to build the demo program. You can also use the makefile as a template for building your own applications that link to the Data Monitor library.

15

Installing Signals

15.1 Introduction	231
15.2 Using the Signal Installation Dialog Box	232
15.3 Installing With the Data Monitor API	241
15.4 Code Instrumentation Alternative	241
15.5 Removing Individual Signals	241
15.6 Process Notes	242

15.1 Introduction

This chapter contains specific information relating to automatically or manually installing signals from a VxWorks or Linux target (see [15.2 Using the Signal Installation Dialog Box](#), p.232). It describes both the **Save** and **Batch Install** options to save and reload, respectively, the collections of signals previously installed using the automatic method. It also addresses using the **Data Monitor API** to install signals (see [15.3 Installing With the Data Monitor API](#), p.241). Finally, this chapter discusses the use of a pre-compiled external execution sampler program (**samplertask.so** for VxWorks, or **processsampler** for Linux), to install all the signals in your target code (see [15.4 Code Instrumentation Alternative](#), p.241).

15.2 Using the Signal Installation Dialog Box

Signals can be installed in a VxWorks or Linux target using the Workbench **Data Monitor Signal Installation** dialog box, in any of the following ways:

- **Automatically**, using a variable name or a variable expression you enter and letting Workbench find the variable and determine its address in the executable.



NOTE: Automatic installation works only on **C**, **C++**, and **assembly** code. For other languages, automatic installation will not work (but manual will).

- **Manually**, by entering the address of the signal.
- **Batch Install**, using the named tab view to reload previously installed signals that have been saved using the **Save** tab view (see [Save Tab View](#), p.239).
- **Instrumenting code**, using the Data Monitor API (see [16.2 Using the Data Monitor API](#), p.248, and [A. API Reference: VxWorks](#) or [B. API Reference: Linux](#) as appropriate).

Although this signal installation method is initiated within Workbench, it is an integral step in using Data Monitor, and is therefore outlined here.

Outside Workbench, you can also install signals using a **sampling process** (see [15.4 Code Instrumentation Alternative](#), p.241).

Recall from the discussion in [Usage Notes](#), p.18, that **installed** signals are the means used by the Data Monitor graphical user interface (GUI) to specify which data you want to be able to monitor, collect, and display in the GUI. You must first install each signal before it can be collected for analysis and displayed. This chapter provides basic guidance for that process for either a VxWorks or Linux target.

The hierarchal steps that must be executed to create installed signals are as follows:

1. **Register a Signal**—Initially you must let Data Monitor know a signal exists by **registering** it using the API call **ScopeRegisterSignal()**. Data Monitor cannot collect data from this signal until it is registered. Registered signals appear in the **Signal Manager** window (see [3.3.7 Signal Manager](#), p.49) in the GUI, where they can be selected for activation.
2. **Activate the signal**—This is done to a registered signal that has been set up on the host by the **Signal Manager** using the API call **ScopeActivateSignal()**. Active signals then appear in the **Signals Bar** of each data-display window (see [Signals Bar](#), p.26).

Once activated, the signal is considered to be **Installed** and is automatically collected from the target, but is not yet displayed in the host GUI until **Selected** in one or more of the four data-display windows: **Plot**, **Plot XY**, **Dump Plot**, and **Monitor**.

3. **Install the signal** (optional)—This is an alternate operation that registers and activates in a single step using the Data Monitor API shortcut **ScopeInstallSignal()** (see *Installing Signals*, p.250). A signal installed in this manner is also seen by the Data Monitor GUI as an installed signal.
4. **Selected signals**—Installed signals are not displayed automatically in the GUI. You must use the GUI to select the installed signals you want to display in the data-display windows, **Plot**, **Plot XY**, **Dump Plot**, and **Monitor**. You can select a different set of signals in each window (see *Signals Bar*, p.26).

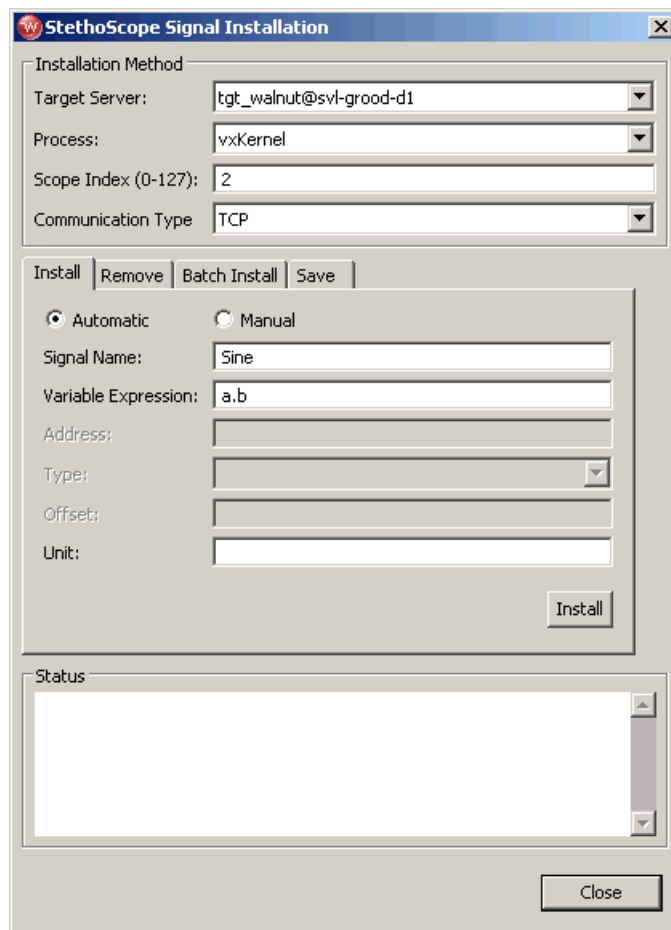
The hierarchical steps listed above can be translated into a simple mechanism for signal installation on a VxWorks or Linux target. These mechanisms use the **Data Monitor Signal Installation** dialog box.



NOTE: A maximum limit of 256 signals can be installed for any given target.

Data Monitor Signal Installation Dialog Box

Open the Workbench **Data Monitor Signal Installation** dialog box with the **Analyze > Data Monitor Signal Installation** command on the Workbench menu bar, or by selecting the **Install signal to Data Monitor** icon on the toolbar.



The top panel, **Installation Method**, contains options you can select based on the target you have connected to. The fields in this panel are:

- **Target Server**

The drop-down list shows the target servers you are currently connected to. Select the target from this list that you wish to install signals to.

- **Process**

The drop down list shows all the processes (kernels or RTPs) running on that target from which you can choose to install signals on.

- **Scope Index (0-127)**

The scope index assigned to the process shown in the field above. You can enter another value if you know what it should be.

- **Communication Type**

Choose the communication type (**TCP** or **WTX**), depending on the type of target server you selected in the first field above. **TCP**, when available, is generally preferable due to its faster speed.

In the center panel of the dialog box there are 4 tab views:

- **Install**
- **Remove**
- **Batch Install**
- **Save**

These tab views are used to enter parameters for the specific signal installation activity you want to invoke. They are covered in detail in the sections that follow.

Install Tab View

The Install tab view (shown in the **Data Monitor Signal Installation** dialog box above) allows you to specify a signal name (and associated parameters) to be installed, either automatically by variable name, or manually by address. The dialog box for automatic installation is shown in the figure above.

The Install tab view contains the following controls.

Parameter Fields:

- **Automatic**

This button, when selected for automatic signal installation, displays the appropriate fields in the tab view. They are:

- **Signal Name**

Specifies the name of the signal to be installed or removed. It does not have to match the **Variable Expression** entry (below). **Signal Name** is optional for signal installation, but it is required to remove a signal or a group of signals.



NOTE: During signal installation, if a signal is of a **class/struct/union** type, the top level directory entry in hierarchical signal names (see [Hierarchical Signal Names](#), p.243) corresponds to the **Signal Name** field if specified or **Variable Expression** field if the **Signal Name** field is blank.

- **Variable Expression**

Specifies a variable name or variable expression that can be used to locate a variable on the target to be installed. Some examples would be **array[3]**, **arm.pos**, and **body->vel**. A Variable expression is not required to remove signals.

- **Unit (Optional)**

Specifies the units of the signal to be installed. Entering a value is optional.

- **Manual**

This button, when selected for manual signal installation, displays different, but appropriate, fields in the **Install** tab view.

The screenshot shows a dialog box with four tabs: 'Install', 'Remove', 'Batch Install', and 'Save'. The 'Install' tab is active. Inside the tab, there are two radio buttons: 'Automatic' (unselected) and 'Manual' (selected). Below the radio buttons are several input fields: 'Signal Name:' (empty), 'Variable Expression:' (empty), 'Address:' (0x00123546), 'Type:' (float *), 'Offset:' (0x00000024), and 'Unit:' (empty). An 'Install' button is located at the bottom right of the dialog.

Manual signal installation is done using this same tab view as automatic signal installation. The difference is that for manual signal installation you must know the virtual address of the signal in the program memory, and provide it as a hexadecimal address to install the signal.

The **Install** tab view entry fields for **Manual** are:

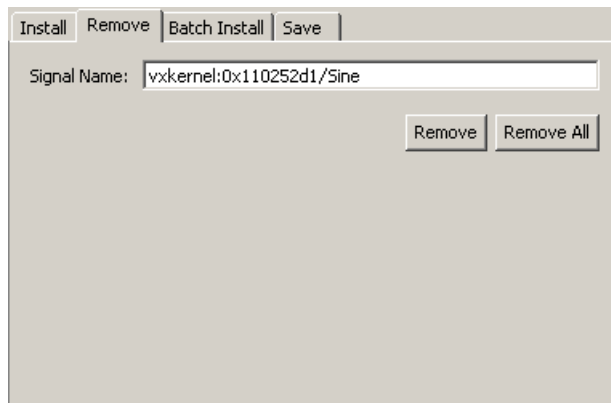
- **Signal Name**

(As described for the Automatic mode above.)

- **Address**
Enter the signal address (in hexadecimal) to be monitored. You must know the address of the variable you want to monitor.
- **Type**
A drop-down menu of allowable data types is available; select one of the types from the menu.
- **Offset**
Enter the offset (in hexadecimal) of a member within a structure, if the variable expression contains a pointer to a structure (or class). You must know the value of this offset.
- **Unit (Optional)**
(As described for the Automatic mode above.)
- **Buttons:**
 - **Install**
Searches for the address and installs it to Data Monitor. The variable expression is used as the signal name if the **Signal Name** field is left blank.

Remove Tab View

Use the **Remove** tab view to specify a signal to be removed (uninstalled).



This tab view contains the following controls:

- **Parameter Fields:**
 - **Signal Name**

The name of the signal you want to remove, with the following caveat: it must be preceded by the full target name, followed by a slash ("/"), for example, **tgt_Walnut/taskIDCurrent**.

- **Buttons:**
 - **Remove**
Searches for the signal name and prefix entered in the **Signal Name** field, and, if found, uninstalls it in Data Monitor.



NOTE: During individual signal **Remove**, only the signal(s) matching the **Signal Name** field entry are removed from the target. Other fields (as described in [Install Tab View](#), p.235 above) are not considered.



CAUTION: For other important information on removing individual signals, see [15.5 Removing Individual Signals](#), p.241.

- **Remove All**
Uninstalls all signals in Data Monitor without having to enter anything in the **Signal Name** field.

Batch Install Tab View

Using the **Batch Install** tab view you can install the list of signals you previously saved (automatically only) using the **Save** tab view (described below).

Name	Expression	Units	

You can re-install the entire list of signals with one click, or install individual signal(s) you select in the list.

This tab view contains the following controls:

▪ **Parameter Fields:**

– **File**

The name of the file you want to batch install from. You can enter it directly, or use the **Browse** button to navigate to a path and existing file name.

– **Signals**

A list of signals contained in the selected file. You can select any or all the signals in the list. All signals selected in the list are installed when you select the **Install** button.

▪ **Buttons:**

– **Install**

Installs the signals selected in the **Signals** list.



CAUTION: Remember, **Batch Install** can only access the list of signals installed up until you reboot your target. If you reboot without first saving the signal list, the list will be lost and you will have to enter them again by hand.

Save Tab View

Using the **Save** tab view you can save the list of signals you previously installed (automatically only) to a file.

Name	Expression	Units

File: Browse

Save

You would want to consider saving your list of signals against the possibility of having to reboot your target (in which case these current signals would otherwise be lost and you would have to enter them again by hand). If you have saved a signal list with this option, you can then load it and re-install the signals at any time using the **Batch Install** tab view of this dialog box.

This tab view contains the following fields:

- **Parameter Fields:**
 - **Signals**
A list of all the signals contained in the file.
 - **File**
The name of the file you want to save the list of signals in. You can enter it directly, or use the **Browse** button to navigate to a path and existing file name.
- **Buttons:**
 - **Save**
Saves the file in the selected directory.



NOTE: The **Save** feature only works for signals installed **automatically** (by variable name) using the **Signal Installation** dialog box. Signals installed **manually** (by address), or with the **Data Monitor API**, cannot be saved.

Status Area

This text area displays the status of your connection to the target agent. When connected, it displays the **host name** and **port number** of the target agent you are connected to, as well as the results of requested actions, and including any error messages.

Buttons

- **Close**
Closes the dialog box.



NOTE: During signal installation, if a signal is of a **class/struct/union** type, the top level directory entry in the hierarchical signal name (see [Hierarchical Signal Names](#), p.243) corresponds to the **Signal Name** field if specified, or **Variable Expression** field if the **Signal Name** field is blank.

15.3 Installing With the Data Monitor API

This method requires you to instrument your target source code with appropriate calls to routines from the Data Monitor API library. You have options with this method, but it requires you to recompile your code in any case. Using this method installs all signals in your target code. The process, and the routines it uses to install and activate signals, are described in detail in [16. API Introduction](#).

For detailed information on just the API routines, see [A. API Reference: VxWorks](#) or [B. API Reference: Linux](#) as appropriate.

15.4 Code Instrumentation Alternative

You may want to take advantage of automatic signal installation, but not always want to instrument your target code with the Data Monitor API library. Or perhaps you may not have the source code available for instrumentation. In such cases there are executable files (**samlertask.so** for VxWorks and **processsampler** for Linux) you can run on your target, that are themselves fully instrumented with the Data Monitor API (see [A. API Reference: VxWorks](#) or [B. API Reference: Linux](#) as appropriate).

This file is found in the directory where you unpacked the target-side components. Connecting to your target will find this sampler task, and you can then use it to automatically install all the signals in your own target code.

15.5 Removing Individual Signals

As mentioned earlier in [Data Monitor Signal Installation Dialog Box](#), p.233, all the installed signals can be removed by simply clicking the **Remove All** button in the **Remove** tab view. However, removing an individual signal from the set of all installed signals is a little more complex. In the **Signal Name** field of this tab view you must fully identify the signal you want to remove according to the origin of the signal. The signal **Sine** (from the demo program output in [2.5 Testing Your](#)

[Installation](#), p.30) is used as an example in [Table 15-1](#), giving the full signal identification, including the prefix attached by the target, for each of 4 possible origins. Use the appropriate entry on the right as a template for your signal name, substituting the indicated prefix values as found in your Signals Tree for everything to the left of the "/" in the template.

Table 15-1

If the signal "Sine" was added to:	...it is identified as:
A VxWorks kernel, where kernelid = 0x12345678	vxkernel:0x12345678/Sine
An RTP named x.vxe with RTP_ID = 0x12345678	x.vxe:0x12345678/Sine
A Linux process with pid = 1234	1234/Sine
Instrumented Linux code	Sine

The example signal **Sine**, used in the table above, happens to be in the top level of the Signals Tree. If your signal is lower down in the tree, be sure to include the entire branch leading up to it, with nodes separated by slashes ("/").

15.6 Process Notes

The following sections clarify and elaborate on the concepts used in the process of Data Monitor signal installation.

Variable Expressions vs. Signal Names

It is important to understand the distinction between **variable expressions** and **signal names**.

- **Variable Expression**

This is the name of a variable that exists in your code and appears in your programs as a data symbol. Valid variable expression entries include:

- Variables of any primitive type, such as **int** or **float**.

- Pointers to any primitive type, such as **int *** or **float ***.
 - Variables of type **enum**, **bool**, or **boolean** are converted to **int** or **char**, depending on how the compiler implemented them.
 - Member variables of structure or class, such as **Body.vel**, **Body->vel**, or **Body->Pos->x**.
 - Array elements, such as **arm[3]**.
 - Instances of **class**, **structure**, or **union**—all member variables (including member variables of **class/struct/union** data types except pointers to the **class/struct/union** types) are installed as separate signals.
 - Array variables, all the array elements are installed.
- **Signal Name**

This is the name you assign to a data item for Data Monitor to display in its signals trees and quick-select buttons. This name is also exported to the files you save. It does not have to be the same as the variable expression. A signal name can be any name you choose.

When using the **Signal Installation** dialog box, only global or static variables may be installed. For example, if the variable expression is:

Body->Pos->x



then **Body** must be a global or static pointer. Also, **Body** must be pointing to a valid **Pos class/struct/union** data pointer type (and if **x** is also a **class/struct/union** data *pointer* type, then it in turn should also point to a valid **class/struct/union**) at the time of installing the signal **x** with the above variable expression.

Hierarchical Signal Names

To help you organize your signals better, Data Monitor supports hierarchical signal names that use the slash (/) character to separate the levels of hierarchy, much like path names for files. Hierarchical organization is useful for:

- Grouping member variables of a class or structure. Just substitute "/" for "." or "->" when entering signal names that refer to member variables.
- Creating logical groupings among variables that are not otherwise in a common structure. Just use a common directory name in their signal names.

Whenever Data Monitor displays signal lists as a tree, the tree consists of signal and directory entries such that:

- A signal entry corresponds to a single registered signal.
- A directory entry (indicated by a  or  node icon) contains sub-entries that are signal entries or other directory entries.
- A directory entry may be expanded or collapsed to show or hide its sub-entries.

Example 15-1 **Examples of hierarchical signal names**

```
Robot/LeftArm/PosX  
Robot/LeftArm/PosY  
Robot/RightArm/PosX  
Robot/RightArm/PosY
```

Classes and Structures

If the named variable refers to an instance of a class, structure, or union, Data Monitor installs all its member variables, grouping them under the same directory (corresponding to the **Signal Name** field or **Variable Expression** field if **Signal Name** field is left blank) using the hierarchical signal-naming capability (see [Hierarchical Signal Names](#), p.243). Data Monitor skips a member variable if it is a pointer to another class, struct, or union to prevent a potentially dangerous circular path of variable installations.

Example 15-2 **Consider the following type definitions and the variable declaration**

```
typedef struct {  
    float PosX;  
    float PosY;  
} Position;          /* Position type definition */  
  
typedef struct {  
    Position RightArm;  
    Position LeftArm;  
} RobotType;         /* RobotType type definition */  
  
RobotType Robot;
```

When you specify a variable expression as **Robot** and signal name as **SCARA** in the **Signal Installation** dialog box, the **Robot** signal is installed with the following hierarchy:

```
SCARA/LeftArm/PosX  
SCARA/LeftArm/PosY  
SCARA/RightArm/PosX  
SCARA/RightArm/PosY
```


If you specify a variable expression, such as **Robot.RightArm** with the **Signal Name** field left blank, **RightArm** member variable is installed as:

```
Robot.RightArm/PosX  
Robot.RightArm/PosY
```

Note that the member variables in a **class/struct/union** are installed in an alphabetical order on the target.

16

API Introduction

- 16.1 Introduction 247
- 16.2 Using the Data Monitor API 248
- 16.3 Understanding Overflows 258
- 16.4 Triggering and Sampling Functions 259
- 16.5 Data Monitor Events API 260
- 16.6 scope.ini File (VxWorks Only) 263

16.1 Introduction

The real-time data-collection and signal-management module of Wind River Data Monitor that runs on the target platform is also known as Data Monitor API. Data Monitor API collects the time history of variables in your real-time program. Its architecture is summarized in [1.2 Architectural Summary](#), p.2.



NOTE: If you are running VxWorks on your target, the Data Monitor API libraries need to be loaded onto the target.



NOTE: If you are running Windows or Solaris on your target, the Data Monitor API library needs to be linked with your application code. For Linux, however, you can use either the Data Monitor API libraries linked to your application code, or you can be running a sampler process (**samlertask.so** for VxWorks and **processsampler** for Linux) which contains an already linked Data Monitor API library.

This chapter introduces the Data Monitor API. For details of the API, see [A. API Reference: VxWorks](#) or [B. API Reference: Linux](#), as appropriate. To compile and run a target application that is instrumented with Data Monitor API already, see [C. Data Monitor Demo Program](#).

16.2 Using the Data Monitor API

The Data Monitor API is a library of routines linked to your target application. It implements a flexible data-collection utility. The Data Monitor API saves data from your real-time system in a buffer on the target and transmits them to the Data Monitor GUI on the host for display.

To use Data Monitor API with your target program, execute the following steps:

1. For VxWorks, initialize the target server (see [Initializing the Target Server \(VxWorks Only\)](#), p.248).
2. Register and activate (install) the variables (signals) to monitor (see [Registering and Activating Signals](#), p.249).
3. Set the sample rate (see [Sampling Signals](#), p.255).
4. Sample the data (see [Sampling Signals](#), p.255).
5. Shut down the server when done.

Initializing the Target Server (VxWorks Only)

Initializing the VxWorks target server requires a single call, **ScopeInitServer()**; you only need to specify the scope index number and buffer sizes. For more details, see [A. API Reference: VxWorks](#).

Scope Index

The scope index represents the communications channel between an instance of Data Monitor API running on the VxWorks target and the Data Monitor GUI running on the host. You can create up to 128 instances of Data Monitor API on a single target machine, each using a different scope index. The index can range from 0 to 127, and it must be specified when you call **ScopeInitServer()**.

Target Buffers

The Data Monitor API allocates three buffers on a VxWorks target for each scope index.

- **Sample buffer**

Stores the data samples for the active signals. Data samples for all data types other than **double** are saved as 4-byte values. Data samples for doubles are saved as 8-byte values.

- **Signal buffer**

Stores the information that describes each registered signal, such as name, units and data type. The memory used by a signal is reclaimed when the signal is removed, making it available for additional signals. The information stored for a signal takes up 36 bytes plus the number of bytes it takes to store the signal name and units (including the terminating null characters). Note that a signal registered twice, under different scope indices, counts as two registered signals.

- **Event buffers**

These are optional buffers that can be attached to an initialized scope index by calling **ScopeEventsAttach**. The event collection APIs (**ScopeEventsCollect** and **ScopeEventMessage**) use these buffers to throw events. There can be a maximum of four event buffers per scope index. Since the event collection APIs are not reentrant, we recommend that each task that throws events use a separate buffer for event collection. The above would obviate the need for mutual exclusion among tasks that employ events.

Registering and Activating Signals

A Data Monitor signal can be any variable in your code of any basic data type, such as **float**, **double**, **unsigned int**, or a pointer to any basic type. [Table 16-1](#) contains a complete list of types and abbreviations.

Table 16-1 **Acceptable Data Types**

unsigned char	uchar	char
unsigned short	ushort	short
unsigned int	uint	int
unsigned long	ulong	long
float	double	



NOTE: Data Monitor supports both the short form and the long form for unsigned types (for example, **uint** vs. **unsigned int**). It also supports user-defined **classes**, **structures**, and **unions**.

For Data Monitor to collect samples of a particular signal, you must follow these steps:

1. **Register** the signal
2. **Activate** the signal

Registering a signal provides Data Monitor support for both the short form and the long form for unsigned types (for example, **uint** vs. **unsigned int**). It also supports user-defined classes, structures, and unions with relevant information such as its name, type, and memory location. Samples are collected, however, only when the signal is activated. The number of registered signals is limited only by the amount of target memory. The number of active signals for a given scope index is, however, limited to 8192.

Installing Signals

A signal is **installed** when it is both **registered** and **activated**. To register and activate a signal, follow these steps:

1. Call **ScopeRegisterSignal()** or **ScopeRegisterSignalWithOffset()** to register the signal. The latter function is discussed in *Offsets to Signals*, p.252.
2. Call **ScopeActivateSignal()** to activate the signal.

You can also call **ScopeInstallSignal()** or **ScopeInstallSignalWithOffset()** to accomplish both these steps in a single call (see *Installing Signals*, p.253).



NOTE: Installed variables must be valid when **ScopeCollectSignals()** is called to sample all active signals. This requirement is met for:

- Static variables (such as, global variables)
- Any variable in allocated memory (that is, created using **malloc()**)
- Automatic (stack) variables, only if **ScopeCollectSignals()** is called only within the scope of the variable.

Example 16-1 Signal Installation

The following code registers and activates signals for scope index of 0:

```
#include "scope/scope.h"
static float PosX;
static float PosY;
static unsigned char Type;
void initScope( void )
{
    ScopeRegisterSignal("PosX", "meters", &PosX,
        "float", 0);
    ScopeRegisterSignal("PosY", "meters", &PosY,
        "float", 0);
    ScopeRegisterSignal("Type", "meters", &Type,
        "uchar", 0);
    ScopeActivateSignal("PosX", 0);
    ScopeActivateSignal("PosY", 0);
    ScopeActivateSignal("Type", 0);
}
```

16

Hierarchical Naming of Signals

Data Monitor supports hierarchical naming of signals just like a file browser. You can register signals using a name that delimits each level with "/", such as **Robot/LeftArm/PosX**. The **Signal Manager** and **Signal Selection** lists within the Data Monitor GUI represent each level as a folder. In order to make it easier to manage a large number of signals, you can expand, collapse, and activate entire directories.

Pointers to Signals

You can register signals by pointers to variables. For these signals, **ScopeCollectSignals()** de-references the pointers at the time of collection. You can register a pointer to a signal by specifying a pointer type when calling **ScopeRegisterSignal()**. For example, a pointer to a **float** would have a type of **float ***. There is no limit to the number of de-references Data Monitor can handle,

so you could even specify a type such as **float *******. Data Monitor supports pointers to all the basic data types listed in [Table 16-1](#).

Example 16-2 **Pointer to Variable Registration**

```
int *JointPosition = (int *)calloc(1, sizeof(int));  
// Notice that you must pass the address of JointPosition.  
ScopeRegisterSignal("JointPosition", "millimeters", &JointPosition,  
                    "int *", 0);
```

Offsets to Signals

You can register a signal using an offset from an address. **ScopeCollectSignals()** de-references the pointers and applies the offsets at the time of collection. This allows you to collect data that is a member variable of a class or structure.

Signal registration using offsets only makes sense when you do have pointers to classes or structures, where the pointers can point to different instances as the application executes. Otherwise you simply could install the member variables directly. Consequently, Data Monitor assumes you are providing a reference to a pointer when you register a signal with offsets.

Example Registration With Offset

To register and activate a signal using an offset, follow these steps:

1. Call **ScopeRegisterSignalWithOffset()**, passing it a reference to a pointer.
2. Call **ScopeActivateSignal()** to activate the signal.

You can combine offsets and pointers. This allows you to collect data from member variables (of structures) that themselves are pointers. For an example, see the next paragraph.

Calculating Offsets

When calling **ScopeRegisterSignalWithOffset()** or **ScopeInstallSignalWithOffset()**, you can use the **offsetof()** macro to calculate automatically the offset of a member variable. The following is an example on how to use **offsetof()**.

Example 16-3 **offsetof () use**

```
typedef struct ArmData_s {
    float      PosX;
    float      PosY;
    unsigned char Type;
    double     *Vel; // Just to demonstrate pointers and offsets.
} ArmData_t;

int main ()
{
    ArmData_t *LeftArmData =
        (ArmData_t *)calloc(1, sizeof(ArmData_t));
    LeftArmData->Vel = (double *)calloc(1, sizeof(double));

    /*
     * Notice that you can pass just the address of LeftArmData,
     * not the individual fields.
     * Notice that we are using hierarchical names for the signals.
     * This will be discussed below.
     * The type "float" refers to the type of
     * LeftArmData->PosX.
     */
    ScopeRegisterSignalWithOffset("LeftArm/PosX", "meters",
                                &LeftArmData, "float",
                                offsetof(ArmData_t, PosX), 0);
    ScopeRegisterSignalWithOffset("LeftArm/PosY", "meters",
                                &LeftArmData, "float",
                                offsetof(ArmData_t, PosY), 0);
    ScopeRegisterSignalWithOffset("LeftArm/Type", "none",
                                &LeftArmData, "uchar",
                                offsetof(ArmData_t, Type), 0);

    /*
     * The type "double *" refers to the type of the member Vel.
     */
    ScopeRegisterSignalWithOffset("LeftArm/Vel", "meters/s",
                                &LeftArmData, "double *",
                                offsetof(ArmData_t, Vel), 0);

    /*
     * Add user code and ScopeCollectSignals(). */
    */
}
```

Installing Signals

Data Monitor provides convenience functions, **ScopeInstallSignal()** and **ScopeInstallSignalWithOffset()**, that register and activate a signal in one step.

Example 16-4 **Installing elements of an array**

```
#include "scope/scope.h"
static float yhat[17];
```

```
ScopeInstallSignal("EstimatorElbowAngle", "radians",  
                  &yhat[2], "float", 0);  
ScopeInstallSignal("EstimatorElbowRate", "rad/sec",  
                  &yhat[3], "float", 0);
```

Example 16-5 Member Variable Installation

```
#include "scope/scope.h"  
typedef struct {  
    int    packetsize;  
    double desiredSwitchValue;  
} *SwitchType;  
SwitchType inputSw;  
  
    inputSw = (SwitchType) malloc(sizeof(*SwitchType));  
ScopeInstallSignal("InputPacketSize", "points",  
                  &inputSw->packetsize, "int", 0);  
ScopeInstallSignal("InputDesiredSwitchValue", "gleebs",  
                  &inputSw->desiredSwitchValue, "double", 0);
```

Deactivating and Removing Signals

You can deactivate and remove signals using the functions:

- **ScopeDeactivateSignal()**
Deactivates a signal, preventing it from being sampled during **ScopeCollectSignals()** calls.
- **ScopeDeactivateMultipleSignals()**
Deactivates signals that match the specified pattern, preventing them from being sampled during **ScopeCollectSignals()** calls.
- **ScopeRemoveSignal()**
Deactivates and unregisters the signal, removing it from the **Signal Manager** of the Data Monitor GUI.
- **ScopeRemoveMultipleSignals()**
Deactivates and unregisters the signals that match the specified pattern, removing them from the **Signal Manager** window of the Data Monitor GUI.

Online Documentation

Consult the reference pages in [16. API Introduction](#) for signal installation and removal functions:

- **ScopeRegisterSignal()**
- **ScopeRegisterSignalWithOffset()**

- **ScopeInstallSignal()**
- **ScopeInstallSignalWithOffset()**
- **ScopeActivateSignal()**
- **ScopeActivateMultipleSignals()**
- **ScopeDeactivateSignal()**
- **ScopeDeactivateMultipleSignals()**
- **ScopeRemoveSignal()**
- **ScopeRemoveMultipleSignals()**
- **ScopeRegisterArray()**
- **ScopeInstallArray()**
- **ScopeShowSignals()**
- **ScopeShowActiveSignals()**

Setting Sample Rate

The sample rate is defined as the frequency of calls to **ScopeCollectSignals()**, for example, the number of times per second your application calls **ScopeCollectSignals()**. Your application determines the sample rate. **ScopeChangeSampleRate()** simply reports that rate to Data Monitor. The sample rate may be changed at any time during execution via another call to **ScopeChangeSampleRate()**.



NOTE: Down-sampling specified by the Data Monitor GUI causes every few calls to **ScopeCollectSignals()** to actually store data.

16

Further details are available under the entries for **ScopeInitServer()** and **ScopeCollectSignals()** in [16.5 Data Monitor Events API](#), p.260.

Sampling Signals

Signals are sampled by calls to the function, **ScopeCollectSignals()**. One sample of each installed variable is taken each time **ScopeCollectSignals()** is called. Your application can call **ScopeCollectSignals()** asynchronously or synchronously with the generation of data.

Asynchronous Sampling

Asynchronous sampling simply takes a snapshot of the variable values at regular intervals during program execution. The intervals are not coordinated with the execution of your application. Because the sampling is independent of the

application task, the snapshots may be taken at unpredictable points in the application code. For example, consider an application with the following loop:

```
void UserTask()
{
    while (1) {
        semTake(sem); /* Wait for something */
        x++;
        y = x;
    }
}
```

One method of asynchronous sampling is to spawn a task whose only job is to sample:

```
void SampleTask()
{
    while (1) {
        ScopeCollectSignals(0);
        taskDelay(delayTime);
    }
}
```

This may produce the following data:

Sample	x	y
1	1	1
2	2	2
3	3	2
4	5	5
5	5	5
...

This result has the following problems:

- Sample 3 appears inconsistent, because the sample was taken between the assignment statements for **x** and **y**.
- Sample 4 is inaccurate because the sampling task missed the fourth loop of the user task altogether.
- Sample 5 is a repeat of Sample 4 because the sampling task ran again before the user task started its next loop.

Thus, asynchronous sampling cannot guarantee:

- **Data-set consistency**

All samples in a set form a consistent view of the application-data set.

- **Sampling accuracy**

Every loop of the application is sampled exactly once.

In spite of the consistency and accuracy issues, asynchronous sampling is often desirable because it is easy to set up and requires no changes to your application code. Asynchronous sampling also works with programs that are not periodic. In many cases all you really need is a general idea of what all your variables are doing; asynchronous sampling does that job well. In fact, you can load and run the demonstration program along with your application and immediately begin viewing your variables. They are sampled asynchronously by the demonstration sample loop.

Synchronous Sampling

Because of the issues with asynchronous sampling, many applications require synchronous sampling, where the sample times must be coordinated with the execution of your application. If your application requires synchronous sampling, call **ScopeCollectSignals()** directly from your application at the instant you wish data to be sampled:

```
void UserTask()
{
    while (1) {
        semTake(sem); /* Wait for something */
        x++;
        y = x;
        ScopeCollectSignals(0);
    }
}
```

This always produces the following consistent, accurate sampling:

Sample	x	y
1	1	1
2	2	2
3	3	3
...

16.3 Understanding Overflows

On the target, calls to **ScopeCollectSignals()** add data samples to the data buffer and the **LinkDaemon** task removes samples from the data buffer to send them to the host.

If the **LinkDaemon** task and host cannot keep up with the rate at which the data buffer is being filled, an overflow will occur when the data buffer is full. An overflow is more likely to occur with a small target data buffer and a large number of active signals.

Overflow Behavior

When the sample buffer on the target overflows, the host is notified immediately. The host then throws away all the samples in its sample buffer (**Buffer Reset**) and clears its plot windows. The string Buffer Reset is displayed in the Mini-Monitor window whenever the sample buffer is cleared. If the Plot window clears frequently accompanied by the appearance of Buffer Reset messages in the Mini-Monitor window, it is a clear case of overflows on the target. Try the recommendations in the following sections to alleviate frequent overflows of the sample buffer.

Avoiding Overflows

You can help prevent buffer overflows by implementing one or both of the following:

- For VxWorks targets, try raising the priority of the **LinkDaemon** task.
- Try increasing the sample buffer size (see the VxWorks **ScopeInitServer()** entry, [ScopeInitServer\(\)](#), p.278, or the Linux **ScopeInitServer()** entry, [ScopeInitServer\(\)](#), p.305).

Notes and Hints

- **Data Collection vs. Buffering**

Do not confuse saving snapshots with data collection. Both stopping data collection (by calling **ScopeCollectionModeDisable()** on the target) and

taking a snapshot freeze the data-displayed in the graphics window. They are, however, quite different actions as explained here.

- If you call **ScopeCollectionModeDisable()**, it prevents further data collection.
- If you take a snapshot, it saves all the data collected in the current cycle, but does not stop data collection. Other data-display windows continue to display the live data.

- **Programmatic Access**

All of the collection control functions can be accessed under program control via calls in the Data Monitor API module on the target. For details, see [13. Using a VxWorks Target](#) or [14. Using a Linux Target](#).

- **If No Data Appears**

No data is collected if data collection is not active. That is, **ScopeCollectSignals()** must be called by the target during each sampling period. Try setting a breakpoint at **ScopeCollectSignals()**.

Alternatively, (on VxWorks only) type the following in the shell:

```
-> b ScopeCollectSignals
```

No data appears (nor would you see an overflow count indicator in the **Data Collection** window) if the **LinkDaemon** task is starved for CPU time, for example, if some higher-priority process is using all of the processor.

- **Live Buffer**

Be sure the Plot window is displaying the live buffer rather than a saved buffer from a previous collection cycle.

16.4 Triggering and Sampling Functions

The triggering and sampling control functions allow run-time access to the data-collection mode and settings from the real-time program. For instance, one of the best uses of the trigger routines is to collect data when some condition is detected by the code. With pre-triggering in effect, this makes it simple to analyze very difficult-to-find error conditions.

The triggering and sampling control functions are:

- `ScopeTriggerSet()`
- `ScopeTriggerGet()`
- `ScopeChangeSampleRate()`
- `ScopeCollectionModeEnable()`
- `ScopeCollectionModeDisable()`
- `ScopeCollectionModeGet()`

For details on these functions, see [ScopeProbe](#), p.268.

16.5 Data Monitor Events API

The Data Monitor Events API is a set of low-overhead logging routines that were made part of Data Monitor Events API version 7.0. These routines can be useful to monitor real-time systems with minimal effect on the timing behavior.

An **event** is defined simply as a line of code represented by a user-specified string (**eventID**). An event is said to be **thrown** when a line of user code instrumented with Data Monitor Events API is executed. Optionally, the value of a variable may also be collected when an event is thrown.

The Data Monitor Events API includes the following routines:

- `ScopeEventsAttach()`
- `ScopeEventsMaskSet()`
- `ScopeEventsCollect()`
- `ScopeEventsMessage()`
- `ScopeEventsDetach()`

Setting Up

To implement the Data Monitor Events API for data collection at strategic locations in your source code, you must first attach an event buffer to an already initialized **scopeIndex** with a call to **ScopeEventsAttach()**. This routine attaches a buffer of the specified size to that **scopeIndex**. A maximum of 4 buffers can be attached to a single **scopeIndex**.



NOTE: Although it is possible to have multiple tasks share a single buffer, we strongly suggest that you only use 1 task per event buffer. This is recommended because the event collection routines, **ScopeEventsCollect()** and **ScopeEventsMessage()**, are not reentrant.

Using

With a collection buffer established, the key to using the Data Monitor Events API is to insert a **ScopeEventsCollect()** statement after each code line you want a message printed for, or that uses a variable you want to collect the value of. **ScopeEventsCollect()** collects, and stores in the buffer, one message, or the value of one global or local variable, per call, without the overhead of using a **printf** statement. The values collected in the events buffer are periodically transmitted to the host by the **LinkDaemon** task running on the target.

A verbosity level (in the range of 1 to 32) can be assigned to each event that is thrown. This allows you to assign different verbosity levels to the messages (or variable values) coming from different modules, and, by using events masks, have control over which ones are collected. Event masks can be set using the **ScopeEventsMaskSet()** routine.

When the task is finished, it should call **ScopeEventsDetach()** to detach the events buffer from the **scopeIndex**. If the **scopeIndex** is shut down, all event buffers associated with it will be freed.

16

Example 16-6 Include Data Monitor Events API in Source Code

```
#include "scope/scope.h"

#define EventLevel1(0x00000001)
#define EventLevel2(0x00000002)
#define EventLevel3(0x00000003)
#define EventLevel30(0x20000000)
#define EventLevel31(0x50000000)
#define EventLevel32(0x80000000)

void ThrowEvents( int scopeIndex )
{
    int eventsHandle = 0;
    int eventsMask = 0;
    double pi = 3.14159;
    int level = 0;
```

```
/* Initialize scopeIndex by calling ScopeInitServer here. */
/* No need to do above if scopeIndex is already initialized. */
/* Attach an event buffer. */
if((eventsHandle = ScopeEventsAttach(-1, scopeIndex)) == 0) {
    printf("Error attaching event buffer to %d!\n",
scopeIndex);
}

/* Set the verbosity mask. */
/* We want events with verbosity 1, 2, 3, 30, 31, and 32. */
eventsMask = (EventLevel1 | EventLevel2 | EventLevel3 | EventLevel30
| EventLevel31 | EventLevel32);

ScopeEventsMaskSet(eventsMask, scopeIndex);

/* Throw events using ScopeEventsCollect and ScopeEventsMessage
calls. */
level = 2;
/* This call will succeed since event level 2 is on. */
ScopeEventsCollect(eventsHandle, level, "Value of pi:", &pi,
"double");

level = 5;
/* This call will not succeed since level 5 is off. */
ScopeEventsCollect(eventsHandle, level, "Value of pi:", &pi,
"double");

level = 32;
/* This call will succeed since level 32 is on. */
ScopeEventsMessage(eventsHandle, level, "The quick brown fox jumps
over the lazy dog.");

level = 25;
/* This call will not succeed since level 25 is off. */
ScopeEventsMessage(eventsHandle, level, "The lazy dog sleeps on top
of the quick brown fox.");

/* Detach the event buffer. */
ScopeEventsDetach(eventsHandle, scopeIndex);
}
```

This example includes all the Data Monitor Events API routines listed at the beginning of this section. It demonstrates how to set up the Data Monitor Events API, how to control the collection of both variable values and messages, and how to properly terminate Data Monitor Events API prior to exiting the task.

Signals vs. Events

The following will help clarify the relationship between **signals** and **events**:

- **Collection**

Signals are **collected** when **ScopeCollectSignals()** is invoked. Events are **thrown** when either **ScopeCollectEvent()** or **ScopeCollectMessage()** is invoked.

- **Frequency of Collection**

Usually, signals are collected periodically. Variables that are persistent (**global/static**) are good candidates for being collected at a certain period.

Variables with limited scope in a program (local variables) can be collected using the Data Monitor Events API. Events are utilized for sporadic collection of ephemeral variables.

- **Timing**

Data Monitor assumes that **ScopeCollectSignals()** was invoked at the sampling rate you set during initialization. The GUI plots each sample with an inter-sample distance proportional to the sampling rate.

Since there is no **period** associated with the Data Monitor Events API, an absolute timestamp is acquired from the target during the time of collection. The Data Monitor GUI then uses this timestamp to place the event at the right temporal location on the **Plot** window.

[7.5 Displaying Events](#), p.130 explains how events are plotted. Though signals and events are different concepts altogether, events can be displayed like signals. Data Monitor joins the samples (collected using **ScopeEventsCollect()**) with lines, to make them appear like signals. It should be noted, however, that for this display method to make sense, the same variable should be collected across multiple collection points.

16.6 scope.ini File (VxWorks Only)

For VxWorks, Data Monitor maintains some global settings in the **scope.ini** file. Most of these settings are maintained by the **Preferences** dialog box (see [3.3.12 Preferences](#), p.53). All default preferences and settings can be restored by deleting **scope.ini** from the same directory where you installed Data Monitor.

Example 16-7 Sample scope.ini File

```
; This section contains some global settings for scope
[defaults]
TotalBufferTime=10
SaveWndPos=0
SaveToolbarPos=0
DontAskWriteback=0
DontAskSave=0

; The following sections save the default window positions
; for the various windows
[SigMan]
wpLeft=1047
wpRight=1461
wpTop=546
wpBottom=938
wpFlags=0
wpShowCmd=1

[LogWnd]
wpLeft=962
wpRight=1518
wpTop=97
wpBottom=595
wpFlags=0
wpShowCmd=1

[PlotWindow]
wpLeft=110
wpRight=1310
wpTop=110
wpBottom=952
wpFlags=0
wpShowCmd=1

[DumpWindow]
wpLeft=303
wpRight=1503
wpTop=233
wpBottom=1094
wpFlags=0
wpShowCmd=1

[PlotXYWindow]
wpLeft=387
wpRight=1087
wpTop=227
wpBottom=927
wpFlags=0
wpShowCmd=1
```

```
[MonitorWindow]
wpLeft=263
wpRight=1463
wpTop=204
wpBottom=1065
wpFlags=0
wpShowCmd=1

; The current colors
; The values are in BGR order
[Colors]
Color0=0x0000ff
Color1=0xff0000
Color2=0xc800c8
Color3=0xc8c800
Color4=0x00c800
Color5=0x4040a0
Color6=0x10bbda4
Color7=0x5a5a5a
Color8=0xc0c0c0
Color9=0x92faf5
Color10=0xfffff00
Color11=0x0099ff
Color12=0x33ff99
Color13=0x330066
Color14=0x990000
Color15=0x08aaa2

; The settings saved for the Preferences Window
[defaults-plot]
YOffset=1.500000
YRange=3.000000
Resolution=1
DispAcc=2
MinGrid=50
MaxSnap=20
SnapToSigs=1
MonRes=1.000000
AllowWrite=1
SelectOnCopy=0
UseSameColors=1
UnselectLive=1
DumpRes=1.000000
DumpHist=1000

[defaults-plotxy]
YOffset=1.500000
YRange=3.000000
XOffset=-1.500000
XRange=3.000000
Resolution=1
DispAcc=2
MinGrid=50
MaxSnap=20
SnapToSigs=1
SelectOnCopy=1
```

```
UseSameColors=1
UnselectLive=1

[defaults-dump]
DispAcc=2
DumpRes=1.000000
DumpHist=500

[defaults-monitor]
DispAcc=4
MonRes=1.000000
AllowWrite=1
```

A

API Reference: VxWorks

`ScopeProbe` – real-time library for Data Monitor 268
`ScopeActivateMultipleSignals()` – activate multiple signals 269
`ScopeActivateSignal()` – activate a signal 269
`ScopeChangeSampleRate()` – change the sampling rate 270
`ScopeCollectSignals()` – collect a sample from each active signal 271
`ScopeCollectionModeDisable()` – disable periodic collection 271
`ScopeCollectionModeEnable()` – enable periodic collection 272
`ScopeCollectionModeGet()` – return the collection mode 272
`ScopeDeactivateMultipleSignals()` – deactivate a group of signals 273
`ScopeDeactivateSignal()` – deactivate a signal 274
`ScopeEventsAttach()` – attach an event buffer to a scope index 274
`ScopeEventsCollect()` – collect the value of a variable on the spot 275
`ScopeEventsDetach()` – detach an event buffer from a scope index 276
`ScopeEventsMaskSet()` – set the events verbosity mask 276
`ScopeEventsMessage()` – throw an event with the specified message 277
`ScopeInitServer()` – initialize a scope index 278
`ScopeInitServerEx()` – initialize a scope index 278
`ScopeInstallArray()` – register and activate an array of signals 280
`ScopeInstallSignal()` – register and activate a signal 281
`ScopeInstallSignalWithOffset()` – register and activate a signal with an offset 282
`ScopePrintVersion()` – print the version number of the Data Monitor target library 284
`ScopeRegisterArray()` – register an array of signals 285
`ScopeRegisterSignal()` – register a signal 286
`ScopeRegisterSignalWithOffset()` – register a signal with an offset 287
`ScopeRemoveMultipleSignals()` – remove several similarly named signals 289
`ScopeRemoveSignal()` – remove a signal 290
`ScopeSampleDivisorSet()` – set the sample divisor for sub-sampling 290
`ScopeShowActiveSignals()` – print all signals that are being collected 291
`ScopeShowSignals()` – print all registered signals 291
`ScopeShutdown()` – shuts down a scope index 291
`ScopeTriggerSet()` – set the triggering parameters 292
`ScopeTriggerGet()` – return the current trigger parameters 293

NOTE: The *scopeIndex* parameter represents the communications **channel** between an instance of Data Monitor API running on the target and a Data Monitor GUI running on the host. You can create up to 128 instances of Data Monitor API on a single target machine, each using a different scope index. The index can range from 0 to 127, and it must be specified when you call **ScopeInitServer()**.

ScopeProbe

NAME	ScopeProbe – real-time library for Data Monitor
ROUTINES	<p>ScopeShutdown() – shut down a scope index</p> <p>ScopeInitServerEx() – initialize a scope index</p> <p>ScopeInitServer() – initialize a scope index</p> <p>ScopePrintVersion() – print the version number of the Data Monitor target</p> <p>ScopeEventsCollect() – collect the value of a variable on the spot</p> <p>ScopeEventsMessage() – throw events with the specified message</p> <p>ScopeEventsMaskSet() – set the event verbosity mask</p> <p>ScopeEventsAttach() – attach an event buffer to a scope index</p> <p>ScopeEventsDetach() – detach an event buffer from a scope index</p> <p>ScopeCollectSignals() – collect a sample from each active signal</p> <p>ScopeCollectionModeEnable() – enable periodic collection</p> <p>ScopeCollectionModeDisable() – disable periodic collection</p> <p>ScopeCollectionModeGet() – return the collection mode</p> <p>ScopeChangeSampleRate() – change the sampling rate</p> <p>ScopeSampleDivisorSet() – set the sample divisor for sub-sampling</p> <p>ScopeRegisterSignalWithOffset() – register a signal with an offset</p> <p>ScopeRegisterSignal() – register a signal</p> <p>ScopeRemoveSignal() – remove a signal</p> <p>ScopeRemoveMultipleSignals() – remove several similarly-named signals</p> <p>ScopeActivateSignal() – activate a signal</p> <p>ScopeActivateMultipleSignals() – activate multiple signals</p> <p>ScopeDeactivateSignal() – deactivate a signal</p> <p>ScopeDeactivateMultipleSignals() – deactivate a group of signals</p> <p>ScopeInstallSignalWithOffset() – register and activate a signal with an</p> <p>ScopeInstallSignal() – register and activate a signal</p> <p>ScopeShowSignals() – print all registered signals</p> <p>ScopeShowActiveSignals() – print all signals that are being collected</p> <p>ScopeRegisterArray() – register an array of signals</p> <p>ScopeInstallArray() – register and activate an array of signals</p> <p>ScopeTriggerSet() – set the triggering parameters</p> <p>ScopeTriggerGet() – return the current trigger parameters</p>

DESCRIPTION This library provides a programmatic interface to Data Monitor real-time data collection and signal management, facilitating collection of time-histories of variables in your real-time program.

ScopeActivateMultipleSignals()

NAME **ScopeActivateMultipleSignals()** – activate multiple signals

SYNOPSIS

```
int ScopeActivateMultipleSignals
(
    const char *namePrefix,
    int        scopeIndex
)
```

DESCRIPTION This function activates signals that have *namePrefix* as prefix. Activated signals can be selected for viewing from one of the many data-display windows (such as **Plot**, **Monitor**, and so forth). **ScopeCollectSignals()** only collects samples of activated signals.

A prefix of "." activates all signals.

RETURNS On success, the number of signals that have been activated.

On failure, it returns 0, indicating one of the following:

- the index *scopeIndex* is invalid or has not been initialized
- there is no signal with the *namePrefix* registered with index *scopeIndex*
- *namePrefix* is NULL or invalid

SEE ALSO **ScopeProbe()**, **ScopeInstallSignal()**, **ScopeInstallSignalWithOffset()**, **ScopeRegisterSignal()**, **ScopeRegisterSignalWithOffset()**, **ScopeActivateSignal()**, **ScopeDeactivateSignal()**, **ScopeRemoveSignal()**, **ScopeRemoveMultipleSignals()**

ScopeActivateSignal()

NAME **ScopeActivateSignal()** – activate a signal

SYNOPSIS

```
RTIBool ScopeActivateSignal
(
    const char .name,
    int        scopeIndex
)
```

DESCRIPTION	This function activates a signal. Activated signals can be selected for viewing from one of the many data-display windows (such as Plot , Monitor , and so forth). ScopeCollectSignals() only collects samples of activated signals.
RETURNS	On success, this function returns RTI_TRUE , indicating that the signal was activated. On failure, it returns RTI_FALSE , indicating one of the following: <ul style="list-style-type: none">▪ the index <i>scopeIndex</i> is invalid or has not been initialized▪ there is no signal named <i>name</i> registered with index <i>scopeIndex</i>
SEE ALSO	ScopeProbe() , ScopeInstallSignal() , ScopeInstallSignalWithOffset() , ScopeRegisterSignal() , ScopeRegisterSignalWithOffset() , ScopeActivateSignal() , ScopeDeactivateSignal() , ScopeRemoveSignal() , ScopeRemoveMultipleSignals()

ScopeChangeSampleRate()

NAME	ScopeChangeSampleRate() – change the sampling rate
SYNOPSIS	<pre>float ScopeChangeSampleRate (float newSampleRate, int scopeIndex)</pre>
DESCRIPTION	<p>This routine changes the amount of time that Data Monitor assumes passed between calls to ScopeCollectSignals(). It does not change the actual sampling rate, that is a responsibility of user code. The rate is used to calculate times between samples. If the rate is incorrect, these calculations will also be incorrect.</p> <p>The parameter should be the frequency in samples per second of the calls to ScopeCollectSignals().</p>
RETURNS	On success, this function returns the old sampling rate. On failure, this function returns 0.0 indicating one of the following: <ul style="list-style-type: none">▪ <i>scopeIndex</i> is invalid or is not initialized▪ <i>rate</i> is invalid (≤ 0.0)
SEE ALSO	ScopeProbe() , ScopeCollectSignals() , ScopeInitServer()

ScopeCollectSignals()

NAME	ScopeCollectSignals() – collect a sample from each active signal
SYNOPSIS	<pre>void ScopeCollectSignals (int scopeIndex)</pre>
DESCRIPTION	<p>This routine should be called periodically to collect the values of signals. Data Monitor will assume that this function is called at the frequency set using ScopeChangeSampleRate(). However, this only affects the timing calculations made by Data Monitor.</p> <p>If a sample divisor is set and is greater than 1, then this routine will return without collecting data when a sample is to be skipped. For instance, with a sample divisor of 3, this routine will only actually collect the data every third time it is called. The other two times, it will simply return immediately.</p> <p>Furthermore, the behavior of this function depends on triggering. If the trigger is set and is waiting for the start condition to occur (ARMED), this function will return without collecting any data. Refer to ScopeTriggerSet() for more information.</p> <p>VxWorks users: the task that calls this routine should have floating point enabled (VX_FP_TASK set).</p> <p>Example code exists under the <i>src</i> directory.</p>
SEE ALSO	ScopeProbe() , ScopeChangeSampleRate() , ScopeCollectionModeEnable() , ScopeCollectionModeDisable() , ScopeTriggerSet()

A

ScopeCollectionModeDisable()

NAME	ScopeCollectionModeDisable() – disable periodic collection
SYNOPSIS	<pre>RTIBool ScopeCollectionModeDisable (int scopeIndex)</pre>
DESCRIPTION	<p>Use this function to disable periodic collection of active signals. Calling this function sets a flag that is examined by ScopeCollectSignals(). If the flag is set, the function returns without sampling active signals. To turn collection on, use ScopeCollectionModeEnable().</p>
RETURNS	On success, this function returns RTI_TRUE .

On failure, this function returns **RTI_FALSE** if:

- *scopeIndex* is invalid or is not initialized

SEE ALSO **ScopeProbe()**, **ScopeCollectSignals()**, **ScopeCollectionModeEnable()**,
ScopeCollectionModeGet()

ScopeCollectionModeEnable()

NAME **ScopeCollectionModeEnable()** – enable periodic collection

SYNOPSIS

```
RTIBool ScopeCollectionModeEnable
(
    int scopeIndex
)
```

DESCRIPTION Use this function to re-enable periodic collection of active signals if it was turned off earlier using **ScopeCollectionModeDisable()**. By default (after calling **ScopeInitServer()**), the collection mode is enabled. Refer to **ScopeCollectionModeDisable()** for more information.

RETURNS On success, this function returns **RTI_TRUE**.
On failure, this function returns **RTI_FALSE** if:
▪ *scopeIndex* is invalid or is not initialized

SEE ALSO **ScopeProbe()**, **ScopeCollectSignals()**, **ScopeCollectionModeDisable()**,
ScopeCollectionModeGet()

ScopeCollectionModeGet()

NAME **ScopeCollectionModeGet()** – return the collection mode

SYNOPSIS

```
int ScopeCollectionModeGet
(
    int scopeIndex
)
```

DESCRIPTION Use this function to determine the current collection mode.

RETURNS On success, this function returns:

- **SCOPE_MODE_ENABLED** if collection mode is enabled
- **SCOPE_MODE_DISABLED** if collection mode is disabled

On failure, this function returns -1 if:

- *scopeIndex* is invalid or is not initialized

SEE ALSO **ScopeProbe(), ScopeCollectSignals(), ScopeCollectionModeEnable(), ScopeCollectionModeDisable()**

ScopeDeactivateMultipleSignals()

NAME **ScopeDeactivateMultipleSignals()** – deactivate a group of signals

SYNOPSIS

```
int ScopeDeactivateMultipleSignals
(
    const char *namePrefix,
    int scopeIndex
)
```

DESCRIPTION This function will remove all signals from the list of active signals for the selected *scopeIndex* which start with the specified *namePrefix*. Therefore, data will not be collected for these signals during **ScopeCollectSignals()**. A prefix of "." deactivates all signals.

RETURNS On success, the number of signals that have been deactivated.

On failure, this function returns 0, indicating one of the following:

- the index *scopeIndex* is invalid or has not been initialized
- there are no active signals with the *namePrefix* registered with index *scopeIndex*
- *namePrefix* is invalid

SEE ALSO **ScopeProbe(), ScopeActivateMultipleSignals()**

ScopeDeactivateSignal()

NAME	ScopeDeactivateSignal() – deactivate a signal
SYNOPSIS	<pre>RTIBool ScopeDeactivateSignal (const char *name, int scopeIndex)</pre>
DESCRIPTION	This function will remove a signal from the list of activate signals. Therefore, data will not be collected for this signal during ScopeCollectSignals() .
RETURNS	On success, this function returns RTI_TRUE , indicating that the signal was deactivated. On failure, this function returns RTI_FALSE , indicating one of the following: <ul style="list-style-type: none">▪ the index <i>scopeIndex</i> is invalid or has not been initialize.▪ there is no active signal named <i>name</i> registered with index <i>scopeIndex</i>
SEE ALSO	ScopeProbe() , ScopeActivateSignal()

ScopeEventsAttach()

NAME	ScopeEventsAttach() – attach an event buffer to a scope index
SYNOPSIS	<pre>int ScopeEventsAttach (int eventBufferSize, int scopeIndex)</pre>
DESCRIPTION	This function attaches an event buffer to a scope index. The eventsHandle returned by this function should be used to throw events. There is a maximum limit of four event buffers that can attach to an index. Throwing events into the same buffer from multiple threads is not recommended.
RETURNS	On success, this function returns a non-zero eventsHandle . On failure, it returns 0 if: <ul style="list-style-type: none">▪ <i>scopeIndex</i> is invalid or is not initialized. The maximum number of event buffers are already attached.

SEE ALSO

ScopeProbe(), **ScopeInitServer()**, **ScopeEventsDetach()**

ScopeEventsCollect()

NAME

ScopeEventsCollect() – collect the value of a variable on the spot

SYNOPSIS

```
void ScopeEventsCollect
(
    int          eventsHandle,
    int          level,
    const char.   *eventId,
    void         ptrToVar,
    RTIAAtomicTypeId typeId
)
```

PARAMETERS

eventsHandle

The handle returned by **ScopeEventsAttach()**.

level

The verbosity level of this event. Range: [1 - 32].

eventId

A *string* that describes this event.

ptrToVar

Pointer to the variable to collect.

typeId

Type identifier for the variable. Use:

- **RTI_INT8ID** for char, unsigned char,
- **RTI_INT16ID** for short, unsigned short,
- **RTI_INT32ID** for int, unsigned int, long, unsigned long,
- **RTI_FLOAT32ID** for float, and
- **RTI_DOUBLE64ID** for double.

DESCRIPTION

This function collects the value of one variable on the spot. The variable does not have to be static/global as required by **ScopeCollectSignals()** and there is no need to perform registration/activation. This function is a low-overhead equivalent of **printf()** for use in debugging realtime systems.

ScopeEventsCollect() is actually a macro that calls the collection function **ScopeEventsCollectInternal()** only if the verbosity level of this event is turned on. Thus, there is no overhead of a function call if the verbosity level of this event is turned off. The verbosity mask can be set using **ScopeEventsMaskSet()**.

NOTE: Calling this function from multiple threads with the same *eventsHandle* is not recommended. Therefore, use a separate *eventsHandle* (returned by **ScopeEventsAttach()**) for each thread, which calls this function.

SEE ALSO **ScopeProbe()**, **ScopeEventsMessage()**, **ScopeEventsAttach()**, **ScopeEventsMaskSet()**

ScopeEventsDetach()

NAME **ScopeEventsDetach()** – detach an event buffer from a scope index

SYNOPSIS

```
RTIBool ScopeEventsDetach
(
    int eventsHandle
)
```

DESCRIPTION *eventsHandle* is the handle returned by **ScopeEventsAttach()**. This function detaches an event buffer from an index.

RETURNS On success, this function returns **RTI_TRUE**.
On failure, it returns **RTI_FALSE** if:

- *eventsHandle* is invalid

SEE ALSO **ScopeProbe()**, **ScopeInitServer()**, **ScopeEventsAttach()**

ScopeEventsMaskSet()

NAME **ScopeEventsMaskSet()** – set the events verbosity mask

SYNOPSIS

```
RTIBool ScopeEventsMaskSet
(
    int eventsMask,
    int scopeIndex
)
```

DESCRIPTION Data Monitor Events API allows for 32 levels of verbosity. This function sets the mask for the specified scope index. Each bit in mask corresponds to one verbosity level. Setting the mask to 0x00000001 turns off all levels except level 1 messages. The mask can be set anytime

during the executing of the program and take effect immediately. By default, all levels are turned on for a scope index.

- RETURNS** On success, this function returns **RTI_TRUE**.
 On failure, this function returns **RTI_FALSE** if:
- *scopeIndex* is invalid or is not initialized
- SEE ALSO** **ScopeProbe()**, **ScopeEventsCollect()**, **ScopeEventsMessage()**, **ScopeEventsAttach()**

ScopeEventsMessage()

NAME **ScopeEventsMessage()** – throw an event with the specified message

SYNOPSIS

```
void ScopeEventsMessage
(
    int eventsHandle,
    int level,
    const char *message
)
```

PARAMETERS *eventsHandle*
 The handle returned by **ScopeEventsAttach()**.

level
 The verbosity level of this event. Range: [1 - 32].

message
 Message string.

DESCRIPTION This function throws an event with the specified message string. This function is a low-overhead equivalent of **printf()** for use in debugging real-time systems.

ScopeEventsMessage() is actually a macro that calls the collection function **ScopeEventsMessageInternal()** only if the verbosity level of this event is turned on. Thus, there is no overhead of a function call if the verbosity level of this event is turned off. The verbosity mask can be set using **ScopeEventsMaskSet()**.

NOTE: Calling this function from multiple threads with the same *eventsHandle* is not recommended. Therefore, use a separate *eventsHandle* (returned by **ScopeEventsAttach()**) for each thread, which calls this function.

SEE ALSO **ScopeProbe()**, **ScopeEventsCollect()**, **ScopeEventsAttach()**, **ScopeEventsMaskSet()**

ScopeInitServer()

NAME	ScopeInitServer() – initialize a scope index
SYNOPSIS	<pre>int ScopeInitServer (int sampleBufferSize, int signalBufferSize, int debugLevel, int scopeIndex)</pre>
DESCRIPTION	Calls ScopeInitServerEx() with default priorities for scopeprobe and scopelink daemon threads. Refer to ScopeInitServerEx() for details.
RETURNS	On success, this function returns the initialized scope index. On failure, this function returns a negative number if: <ul style="list-style-type: none">▪ <i>scopeIndex</i> is out of range▪ all indices are occupied▪ memory allocation failed: it failed to create daemon threads (which will happen if the threads failed to bind to TCP ports, or if the priorities specified are not valid on the target operating system)
SEE ALSO	ScopeProbe() , ScopeInitServerEx()

ScopeInitServerEx()

NAME	ScopeInitServerEx() – initialize a scope index
SYNOPSIS	<pre>int ScopeInitServerEx (int sampleBufferSize, int signalBufferSize, int debugLevel, int probeDaemonPriority, int linkDaemonPriority, int scopeIndex)</pre>
DESCRIPTION	Initializes the scope index <i>scopeIndex</i> . If <i>scopeIndex</i> is -1, then this function initializes the next uninitialized scope index. This should be called once for each index (on VxWorks, normally at boot time). Calling it multiple times for the same index is harmless though.

The *sampleBufferSize* parameter specifies the size of the target data buffer in bytes. This buffer is used to store the data samples for the active signals. Data samples for all types other than **double** are saved as 4-byte values. Data samples for doubles are saved as 8-byte values. Thus, the maximum number of samples that can fit in the buffer will range between $((\text{sampleBufferSize} / 8) - 1) / \text{numberOfSignals}$ and $((\text{sampleBufferSize} / 4) - 1) / \text{numberOfSignals}$, rounded down to the nearest integer.

If *sampleBufferSize* ≤ 0 , it defaults to (32×1024) . Otherwise, if it is less than 1024, a value of 1024 is used instead.

The *signalBufSize* parameter specifies the size of the target signal buffer in bytes. This buffer is used to store the information specific to each signal (such as the name, units and type). The space taken by a signal gets reclaimed (for registering other signals) when the signal is removed. The information stored for a signal takes up 28 bytes plus the number of bytes (including the terminating null character) it takes to store the signal name and units. Note that a signal registered twice, under different scope indices, counts as two registered signals.

If *signalBufferSize* ≤ 0 , it defaults to (32×1024) . If *signalBufferSize* > 0 and < 1024 it defaults to 1024.

The *scopeprobeDaemonPriority* and *scopelinkDaemonPriority* parameters specify the scheduling priority levels for the **scopeprobe** and **scopelink** daemon threads that are spawned by this function. Refer to the documentation on threads and scheduling for the operating system that you are using, for valid priority levels. If you do not want to specify a priority level, you can pass a nonpositive value for this parameter. In that case, suitable priority levels will automatically be chosen.

Normally, for each scope index, a target spawns real-time daemons (**scopeprobe** and **scopelink**) which communicate to the host over the network, using TCP/IP. If you are using Tornado, and wish to use the Tornado WTX protocol instead, you must load the Data Monitor WTX target library (**libscopewtx.so**). In that case, the daemons will use the WTX protocol to communicate even if IP is available. Note that the protocol is called **WTX**, even though the target is talking to the WDB daemon.

RETURNS

On success, this function returns the initialized scope index.

On failure, this function returns a negative number if:

- *scopeIndex* is out of range
- all indices are occupied
- memory allocation failed: it failed to create daemon threads (which will happen if the threads failed to bind to TCP ports, or if the priorities specified are not valid on the target operating system)

SEE ALSO

ScopeProbe(), ScopeShutdown(), ScopeEventsAttach(), ScopeEventsDetach()

ScopeInstallArray()

NAME	ScopeInstallArray() – register and activate an array of signals
SYNOPSIS	<pre>int ScopeInstallArray (const char *name, int elementsInArray, const char *units, void .ptrToStaticArray, const char *type, int scopeIndex)</pre>
PARAMETERS	<p><i>name</i> A unique name to identify the array.</p> <p><i>elementsInArray</i> The number of elements to install.</p> <p><i>units</i> User specified unit of measurement, for identification.</p> <p><i>ptrToStaticArray</i> Must be a pointer to a static (or global) storage location.</p> <p><i>type</i> Must be one of the accepted types such as:</p> <ul style="list-style-type: none">– <i>double</i>– <i>float</i>– <i>int</i> <p>and so forth.</p> <p>See ScopeProbe() for details on types.</p>
DESCRIPTION	This function calls ScopeRegisterArray() followed by ScopeActivateMultipleSignals() . It exists purely for convenience. Users should refer to ScopeRegisterArray() and ScopeActivateMultipleSignals() for details.
EXAMPLES	<pre>float global_array[100]; int main() { static double static_array[50]; // Initialize ScopeIndex here. See ScopeInitServer(). ScopeInstallArray("my_global_array1", 100, "none", &global_array, "float", ScopeIndex); ScopeInstallArray("my_static_array1", 50, "none", &static_array,</pre>

```
        "double", ScopeIndex);
while ( ) {
    ScopeCollectSignals( );
}
// Shutdown ScopeIndex. See ScopeShutdown( ).
}
```

RETURNS On success, this function returns the numbers of array elements successfully installed.
 On failure, it returns zero, indicating one of the following:

- the index (*scopeIndex*) is invalid or has not been initialized
- the type specified (*type*) is not recognized
- the parameters passed are invalid
- there is no more space in the signal buffer

SEE ALSO [ScopeProbe\(\)](#), [ScopeRegisterArray\(\)](#), [ScopeActivateMultipleSignals\(\)](#),
[ScopeDeactivateMultipleSignals\(\)](#), [ScopeInstallSignal\(\)](#)

ScopeInstallSignal()

NAME [ScopeInstallSignal\(\)](#) – register and activate a signal

SYNOPSIS

```
RTIBool ScopeInstallSignal
(
    const char *name,
    const char *units,
    void .ptrToStaticVar,
    const char *type,
    int scopeIndex
)
```

DESCRIPTION This function calls [ScopeRegisterSignal\(\)](#) followed by [ScopeActivateSignal\(\)](#). It exists purely for backwards compatibility and convenience. Users should refer to [ScopeRegisterSignal\(\)](#) and [ScopeActivateSignal\(\)](#) for details.

EXAMPLES

```
float global_var;
float .global_ptr;

int main( )
{
    static double static_var;
    static double .static_ptr;
```

```
// Initialize ScopeIndex here. See ScopeInitServer( ).
ScopeInstallSignal("my_global_var1", "none", &global_var, "float",
                  ScopeIndex);
ScopeInstallSignal("my_static_var1", "none", &static_var,
                  "double", ScopeIndex);
global_ptr = (float .) calloc(1, sizeof(global_ptr));
static_ptr = (double .) calloc(1, sizeof(static_ptr));
ScopeInstallSignal("my_global_var2", "none", global_ptr, "double",
                  ScopeIndex);
ScopeInstallSignal("my_static_var2", "none", static_ptr, "double",
                  ScopeIndex);
while ( ) {
    ScopeCollectSignals( );
}
// Shutdown ScopeIndex. See ScopeShutdown( ).
}
```

RETURNS On success, this function returns **RTI_TRUE**, indicating that the signal was installed.

On failure, it returns **RTI_FALSE**, indicating one of the following:

- the index (*scopeIndex*) is invalid or has not been initialized
- the type specified (*type*) is not recognized
- the parameters passed are invalid
- there is no more space in the signal buffer

SEE ALSO [ScopeProbe\(\)](#), [ScopeRegisterSignal\(\)](#), [ScopeActivateSignal\(\)](#), [ScopeDeactivateSignal\(\)](#), [ScopeRemoveSignal\(\)](#), [ScopeRemoveMultipleSignals\(\)](#)

ScopeInstallSignalWithOffset()

NAME [ScopeInstallSignalWithOffset\(\)](#) – register and activate a signal with an offset

SYNOPSIS

```
RTIBool ScopeInstallSignalWithOffset
(
    const char *name,
    const char *units,
    void_ptrToStaticVar,
    const char *type, int offset,
    int scopeIndex
)
```

DESCRIPTION Register and activate a signal with an offset.

PARAMETERS *name*
A unique name to identify the signal.

units

User specified unit of measurement, for identification.

ptrToStaticVar

Must point to a static (or global) storage location.

type

Refers to the type of the variable at the given *offset*.

offset

Number of bytes from *ptrToStaticVar* to collect data from.

DESCRIPTION

This function calls **ScopeRegisterSignalWithOffset()** followed by **ScopeActivateSignal()**. It exists purely for convenience. Users should refer to **ScopeRegisterSignalWithOffset()** and **ScopeActivateSignal()** for details.

EXAMPLES

The following illustrates how to use **ScopeInstallSignalWithOffset()**:

```
typedef struct ArmData_s {
    float PosX;
    float PosY;
    unsigned char Type;
    double .Vel; // Just to demonstrate pointers and offsets.
} ArmData_t;

int main ( )
{
    ArmData_t .LeftArmData = (ArmData_t.)calloc(1, sizeof( ));
    LeftArmData->Vel = (double.)calloc(1, sizeof( ));

    // Initialize ScopeIndex here. See ScopeInitServer( ).
    // Notice that you can pass just the address of LeftArmData,
    // not the individual fields.
    // The type "float" refers to the type of LeftArmData->PosX.
    ScopeInstallSignalWithOffset("LeftArm/PosX", "meters",
                                &LeftArmData, "float",
                                offsetof(ArmData_t, PosX), 0);

    ScopeInstallSignalWithOffset("LeftArm/PosY", "meters",
                                &LeftArmData, "float",
                                offsetof(ArmData_t, PosY), 0);

    ScopeInstallSignalWithOffset("LeftArm/Type", "meters",
                                &LeftArmData, "uchar",
                                offsetof(ArmData_t, Type), 0);

    // The type "double ." refers to the type of the member Vel.
    ScopeInstallSignalWithOffset("LeftArm/Vel", "meters", &LeftArmData,
                                "double .", offsetof(ArmData_t, Vel), 0);
```

```
while ( ) {  
    // Set the data in the member LeftArmData.  
    ScopeCollectSignals( );  
}  
  
// Shutdown ScopeIndex. See ScopeShutdown( ).  
}
```

RETURNS	On success, this function returns RTI_TRUE , indicating that the signal was installed. On failure, it returns RTI_FALSE , indicating one of the following: <ul style="list-style-type: none">▪ the index (<i>scopeIndex</i>) is invalid or has not been initialized▪ the type specified (<i>type</i>) is not recognized▪ the parameters passed are invalid▪ there is no more space in the signal buffer
SEE ALSO	ScopeProbe(), ScopeRegisterSignal(), ScopeActivateSignal(), ScopeDeactivateSignal(), ScopeRemoveSignal(), ScopeRemoveMultipleSignals()

ScopePrintVersion()

NAME	ScopePrintVersion() – print the version number of the Data Monitor target library
SYNOPSIS	<code>void ScopePrintVersion(void)</code>
DESCRIPTION	Print the version number of the Data Monitor target library.
SEE ALSO	ScopeProbe(), ScopeProbe()

ScopeRegisterArray()

NAME **ScopeRegisterArray()** – register an array of signals

SYNOPSIS

```
int ScopeRegisterArray
(
    const char *name,
    int elementsInArray,
    const char *units,
    void .ptrToStaticArray,
    const char *type,
    int scopeIndex
)
```

DESCRIPTION This function registers an array (one dimensional) of signals by calling **ScopeRegisterSignal()** for each element in the array. An array modifier "[I]" is appended to the signal name followed by the index of the element.

A valid signal name should not contain asterisks or blank spaces. Invalid characters in signal name (*name*) would be automatically replaced with underscores.

It is the responsibility of teh caller to ensure that the address passed as *ptrToStaticArray* points to a valid memory location. In particular, a bad address may cause a **Bus Error** or **Segmentation Fault** to occur within **ScopeCollectSignals()**.

EXAMPLES

```
float global_array[100];

int main( )
{
    static double static_array[50];

    // Initialize ScopeIndex here. See ScopeInitServer( ).
    ScopeRegisterArray("my_global_array1", 100, "none", &global_array,
        "float", ScopeIndex);
    ScopeRegisterArray("my_static_array1", 50, "none", &static_array,
        "double", ScopeIndex);

    ScopeActivateMultipleSignals("my_global_array1", ScopeIndex);
    ScopeActivateMultipleSignals("my_static_array1", ScopeIndex);

    while ( ) {
        ScopeCollectSignals( );
    }
    // Shutdown ScopeIndex. See ScopeShutdown( ).
}
```

RETURNS On success, this function returns the numbers of array elements successfully registered.

On failure, it returns zero, indicating one of the following:

- the index *scopeIndex* is invalid or has not been initialized

- the type specified *type* is not recognized
- the parameters passed are invalid
- there is no more space in the signal buffer

SEE ALSO

`ScopeProbe()`, `ScopeRegisterSignal()`, `ScopeInstallArray()`,
`ScopeActivateMultipleSignals()`, `ScopeDeactivateMultipleSignals()`

ScopeRegisterSignal()

NAME

`ScopeRegisterSignal()` – register a signal

SYNOPSIS

```
RTIBool ScopeRegisterSignal
(
    const char *name,
    const char *units,
    void .ptrToStaticVar,
    const char *type,
    int scopeIndex
)
```

DESCRIPTION

This function registers a *signal*. A registered signal is one that is made known to the Data Monitor signal manager, but not to any other window, for example, the **Plot** window. It has the possibility of becoming **activated** through the signal manager or through the target function `ScopeActivateSignal()`. Activated signals can be selected for display from one of the many data-display windows (for example, **Plot**, **Monitor**, and so forth).

A signal is any variable in the program, with the caveat that it must have a valid value at the instant that `ScopeCollectSignals()` is called. Thus, most anything can be installed as a signal, with the exception of automatic stack variables whose scope does not include the `ScopeCollectSignals()` call.

A valid signal name should not contain asterisks or blank spaces. Invalid characters in signal *name* would be automatically replaced with underscores.

It is the responsibility of the caller to ensure that the address passed *ptrToStaticVar* points to a valid memory location. In particular, a bad address may cause a **Bus Error** or **Segmentation Fault** to occur within `ScopeCollectSignals()`.

EXAMPLES

```
float global_var;
float .global_ptr;

int main( )
{
    static double static_var;
    static double .static_ptr;
```

```
// Initialize ScopeIndex here. See ScopeInitServer( ).

ScopeRegisterSignal("my_global_var1", "none", &global_var,
                   "float", ScopeIndex);
ScopeRegisterSignal("my_static_var1", "none", &static_var,
                   "double", ScopeIndex);

global_ptr = (float .) calloc(1, sizeof(.global_ptr));
static_ptr = (double .) calloc(1, sizeof(.static_ptr));

ScopeRegisterSignal("my_global_var2", "none", global_ptr,
                   "double", ScopeIndex);
ScopeRegisterSignal("my_static_var2", "none", static_ptr,
                   "double", ScopeIndex);

// Activate the registered signals here. See ScopeActivateSignal( ).

while ( ) {
    ScopeCollectSignals( );
}

// Shutdown ScopeIndex. See ScopeShutdown( ).
}
```

RETURNS On success, this function returns **RTI_TRUE**, indicating that the signal was registered.

On failure, it returns **RTI_FALSE**, indicating one of the following:

- the index *scopeIndex* is invalid or has not been initialized
- the type specified *type* is not recognized
- the parameters passed are invalid
- there is no more space in the signal buffer

SEE ALSO **ScopeProbe(), ScopeActivateSignal(), ScopeDeactivateSignal(), ScopeInstallSignal(), ScopeRegisterSignalWithOffset()**

ScopeRegisterSignalWithOffset()

NAME **ScopeRegisterSignalWithOffset()** – register a signal with an offset

SYNOPSIS

```
RTIBool ScopeRegisterSignalWithOffset
(
    const char *name,
    const char *units,
    void .ptrToStaticVar,
    const char *type,
    int offset,
    int scopeIndex
)
```

DESCRIPTION

This function has the same functionality as **ScopeRegisterSignal()**, except callers can specify an offset from *ptrToStaticVar*. This offset refers to a variable in a structure that a caller wants to sample. However, the address of the variable may not be known at register time, hence the offset.

A valid signal name should not contain asterisks or blank spaces. Invalid characters in signal *name* would be automatically replaced with underscores.

It is the responsibility of the caller to ensure that the address and offset parameters point to a valid memory location. In particular, a bad address may cause a **Bus Error** or **Segmentation Fault** during **ScopeCollectSignals()**.

Ideally, this function would be used to sample variables in an array of structures. The caller would register a structure pointer with an offset to a variable they want sampled. Therefore, by changing the pointer callers can sample the same variable at different locations in the array, see below for examples.

EXAMPLES

```
// An example structure.
typedef struct TestData_s {
    float field1;
    double field2;
} TestData_t;

// Here is an array that is read in.
#define MAXLENGTH 100
TestData_t TestDataArray[MAXLENGTH];
TestData_t .TestDataPtr = &TestDataArray[0];

int main( )
{
    int i = 0;

    // Initialize ScopeIndex here. See ScopeInitServer( ).

    // Notice we are using the new hierarchical naming feature.
    ScopeRegisterSignalWithOffset("test/field1", "volts",
                                &TestDataPtr, "float",
                                offsetof(TestData_t, field1),
                                ScopeIndex);
    ScopeRegisterSignalWithOffset("test/field2", "volts",
                                &TestDataPtr, "double",
                                offsetof(TestData_t, field2),
                                ScopeIndex);

    // Activate the registered signals here. See ScopeActivateSignal( ).

    for (i = 0; i < MAXLENGTH; i++) {
        // Read in TestDataArray[i].
        // Set TestDataPtr to the new data.
        TestDataPtr = &TestDataArray[i];
        ScopeCollectSignals( );
    }
}
```

```
// Shutdown ScopeIndex. See ScopeShutdown( ).
}
```

RETURNS On success, this function returns **RTI_TRUE**, indicating that the signal was registered.
On failure, it returns **RTI_FALSE**, indicating one of the following:

- the index *scopeIndex* is invalid or has not been initialized
- the type specified *type* is not recognized
- the parameters passed are invalid
- there is no more space in the signal buffer

SEE ALSO **ScopeProbe()**, **ScopeRegisterSignal()**, **ScopeActivateSignal()**, **ScopeInstallSignal()**, **ScopeInstallSignalWithOffset()**, **ScopeRemoveSignal()**, **ScopeRemoveMultipleSignals()**, **ScopeDeactivateSignal()**

ScopeRemoveMultipleSignals()

NAME **ScopeRemoveMultipleSignals()** – remove several similarly named signals

SYNOPSIS

```
int ScopeRemoveMultipleSignals
(
    const char *namePrefix,
    int scopeIndex
)
```

DESCRIPTION This function removes a set of installed signals that have *namePrefix* as prefix. If *namePrefix* is ".", all signals will be removed.

RETURNS On success, returns the number of signals removed.
On failure, this function returns 0, indicating one of the following:

- *namePrefix* is NULL
- there is no signal named with prefix *namePrefix* registered with index *scopeIndex*
- the index *scopeIndex* is out of range or if index is not initialized

SEE ALSO **ScopeProbe()**, **ScopeRemoveSignal()**

ScopeRemoveSignal()

NAME	ScopeRemoveSignal() – remove a signal
SYNOPSIS	<pre>RTIBool ScopeRemoveSignal (const char *name, int scopeIndex)</pre>
DESCRIPTION	This function deactivates and unregisters a signal. This will invalidate any partially filled collection buffer. After this call users can not activate this signal.
RETURNS	<p>On success, this function returns RTI_TRUE, indicating that the installed signal was removed.</p> <p>On failure, it returns RTI_FALSE, indicating one of the following:</p> <ul style="list-style-type: none">▪ the index <i>scopeIndex</i> is invalid or has not been initialized▪ there is no signal named <i>name</i> registered with index <i>scopeIndex</i>
SEE ALSO	ScopeProbe() , ScopeInstallSignal() , ScopeInstallSignalWithOffset() , ScopeRegisterSignal()

ScopeSampleDivisorSet()

NAME	ScopeSampleDivisorSet() – set the sample divisor for sub-sampling
SYNOPSIS	<pre>int ScopeSampleDivisorSet (int newSampleDivisor, int scopeIndex)</pre>
DESCRIPTION	This function sets the sample divisor for this index. From then on, ScopeCollectSignals() will succeed only once in <i>newSampleDivisor</i> number of times it is invoked.
RETURNS	<p>On success, this function returns the old sample divisor.</p> <p>On failure, this function returns 0 indicating one of the following:</p> <ul style="list-style-type: none">▪ <i>scopeIndex</i> is invalid or is not initialized▪ <i>newSampleDivisor</i> is invalid (Range: 1 - 10,000)

SEE ALSO `ScopeProbe()`, `ScopeCollectSignals()`, `ScopeInitServer()`

ScopeShowActiveSignals()

NAME `ScopeShowActiveSignals()` – print all signals that are being collected

SYNOPSIS `void ScopeShowActiveSignals(int scopeIndex)`

DESCRIPTION This function prints a formatted list of all currently installed signals to the standard output. The current signal value is also printed.

SEE ALSO `ScopeProbe()`, `ScopeInstallSignal()`, `ScopeInstallSignalWithOffset()`,
`ScopeRegisterSignal()`, `ScopeRegisterSignalWithOffset()`

ScopeShowSignals()

NAME `ScopeShowSignals()` – print all registered signals

SYNOPSIS `void ScopeShowSignals(int scopeIndex)`

DESCRIPTION This function prints a formatted list of all currently registered signals to the standard output. The current signal value is also printed.

SEE ALSO `ScopeProbe()`, `ScopeInstallSignal()`, `ScopeInstallSignalWithOffset()`,
`ScopeRegisterSignal()`, `ScopeRegisterSignalWithOffset()`

ScopeShutdown()

NAME `ScopeShutdown()` – shuts down a scope index

SYNOPSIS `RTIBool ScopeShutdown`
 `(`
 `int scopeIndex`
 `)`

DESCRIPTION	This function shuts down the scope index <i>scopeIndex</i> . Event buffers, if any, associated with this index are also shutdown.
RETURNS	On success, this function returns RTI_TRUE . On failure, this function returns RTI_FALSE if: <ul style="list-style-type: none">▪ <i>scopeIndex</i> is invalid or not initialized
SEE ALSO	ScopeProbe() , ScopeInitServer() , ScopeInitServerEx() , ScopeEventsAttach() , ScopeEventsDetach()

ScopeTriggerSet()

NAME	ScopeTriggerSet() – set the triggering parameters
SYNOPSIS	<pre>RTIBool ScopeTriggerSet (ScopeTriggerInfoPtr pTriggerStart, ScopeTriggerInfoPtr pTriggerStop, RTIBool triggerRearm, int scopeIndex)</pre>
DESCRIPTION	<p>This function sets up a trigger. Triggering is a way to control when and for how long periodic data (collected using ScopeCollectSignals()) is collected. A trigger consists of a start condition and a stop condition. After the trigger is armed using ScopeTriggerSet(), all calls to ScopeCollectSignals() will return without collecting data. Data collection will resume only after the specified start condition is met. The start condition can be based on a signal or an event or a trigger can be set with no start condition (trigger immediately option). The stop condition can be based on a signal or an event. Stop condition for a trigger based on a set time period is available only from the Data Monitor GUI.</p> <p>The trigger will expire as soon as the stop condition is met. Data collection will continue as before after the trigger expires. If the <i>triggerRearm</i> flag is set, the trigger will again be set with the values passed.</p> <p>To set a trigger, fill out the following fields in the ScopeTriggerInfo() structure:</p>

source

The source of the condition. Can be one of:

- **SCOPE_TRIG_SRC_SIGNAL** (triggers on a signal)
- **SCOPE_TRIG_SRC_EVENT** (triggers on an event)
- **SCOPE_TRIG_SRC_NOW** (triggers immediately)

slope

The slope of the signal as it passes the level in order to trigger:

- **SCOPE_TRIG_SLOPE_POS** (positive slope)
- **SCOPE_TRIG_SLOPE_NEG** (negative slope)
- **SCOPE_TRIG_SLOPE_ANY** (any slope)

This parameter is relevant only if the source is a signal.

level

A value that the signal must reach in order to trigger. This parameter is relevant only if the source is a signal.

signal

The name of the signal to trigger on.

eventId

The event to trigger on.

To set a trigger based on a signal, fill out *source*, *slope*, *level*, and *signal*. To set a trigger based on an event, fill out the *eventId*. To set the trigger to start immediately, fill out *source* (set it to **SCOPE_TRIG_SRC_NOW**).

For all the trigger modes listed above, pass in a non-NULL value to parameters that are not used in that mode.

To disable the trigger, pass NULL for **pTriggerStart** and **pTriggerStop**.

RETURNS

On success, this function returns **RTI_TRUE**, indicating that the trigger was set.

On failure, this function returns **RTI_FALSE** if:

- *scopeIndex* is invalid or is not initialized

SEE ALSO

ScopeProbe(), **ScopeTriggerGet()**, **ScopeCollectSignals()**

ScopeTriggerGet()

NAME

ScopeTriggerGet() – return the current trigger parameters

SYNOPSIS

```
RTIBool ScopeTriggerGet
(
    int .pTriggerMode,
    int .pTriggerRearm,
    ScopeTriggerInfoPtr pTriggerStart,
    ScopeTriggerInfoPtr pTriggerStop,
    int scopeIndex
)
```

DESCRIPTION	<p>This function returns the current trigger parameters.</p> <p>If pTriggerMode returns:</p> <ul style="list-style-type: none">▪ SCOPE_TRIG_MODE_DISABLED, then there is no trigger set for that index.▪ SCOPE_TRIG_MODE_ARMED, there is a trigger set for that index and is waiting for the start condition to occur. This also means that no data is being collected by ScopeCollectSignals().▪ SCOPE_TRIG_MODE_TRIGGERING, there is a trigger set for that index and is waiting for the stop condition to occur.
RETURNS	<p>On success, this function returns RTI_TRUE.</p> <p>On failure, this function returns RTI_FALSE if:</p> <ul style="list-style-type: none">▪ <i>scopeIndex</i> is invalid or has not been initialized
SEE ALSO	<p>ScopeProbe(), ScopeTriggerSet(), ScopeCollectSignals()</p> <p>ScopeCollectSignals() – collect a sample from each active signal 270</p>

B

API Reference: Linux

`ScopeProbe()` – real-time library for Data Monitor 296
`ScopeActivateMultipleSignals()` – activate multiple signals 297
`ScopeActivateSignal()` – activate a signal 297
`ScopeChangeSampleRate()` – change the sampling rate 298
`ScopeCollectSignals()` – collect a sample from each active signal 299
`ScopeCollectionModeDisable()` – disable periodic collection 299
`ScopeCollectionModeEnable()` – enables periodic collection. 300
`ScopeCollectionModeGet()` – return the collection mode 300
`ScopeDeactivateMultipleSignals()` – deactivate a group of signals 301
`ScopeDeactivateSignal()` – deactivate a signal 302
`ScopeEventsAttach()` – attach an event buffer to a scope index 302
`ScopeEventsCollect()` – collect the value of a variable on the spot 303
`ScopeEventsDetach()` – detach an event buffer from a scope index 303
`ScopeEventsMaskSet()` – set the events verbosity mask 304
`ScopeEventsMessage()` – throw an event with the specified message 304
`ScopeInitServer()` – initialize a scope index 305
`ScopeInitServerEx()` – initialize a scope index 306
`ScopeInstallArray()` – register and activate an array of signals 307
`ScopeInstallSignal()` – register and activate a signal 308
`ScopeInstallSignalWithOffset()` – register and activate a signal with an offset 310
`ScopePrintVersion()` – print the version number of the Data Monitor target library 311
`ScopeRegisterArray()` – register an array of signals 312
`ScopeRegisterSignal()` – register a signal 313
`ScopeRegisterSignalWithOffset()` – register a signal with an offset 315
`ScopeRemoveMultipleSignals()` – remove several similarly-named signals 316
`ScopeRemoveSignal()` – remove a signal 317
`ScopeSampleDivisorSet()` – set the sample divisor for sub-sampling 318
`ScopeShowActiveSignals()` – print all signals that are being collected 318
`ScopeShowSignals()` – print all registered signals. 319
`ScopeShutdown()` – shut down a scope index 319
`ScopeTriggerGet()` – return the current trigger parameters 320
`ScopeTriggerSet()` – set the triggering parameters 321

NOTE: The *scopeIndex* parameter represents the communications **channel** between an instance of Data Monitor API running on the target and a Data Monitor GUI running on the host. You can create up to 128 instances of Data Monitor API on a single target machine, each using a different scope index. The index can range from 0 to 127, and it must be specified when you call **ScopeInitServer()**.

ScopeProbe()

NAME ScopeProbe() – real-time library for Data Monitor

SYNOPSIS

ScopeShutdown() - shut down a scope index
ScopeInitServerEx() - initialize a scope index
ScopeInitServer() - initialize a scope index
ScopePrintVersion() - print the version number of the Data Monitor target
ScopeEventsCollect() - collect the value of a variable on the spot
ScopeEventsMessage() - throw an event with the specified message
ScopeEventsMaskSet() - set the event verbosity mask
ScopeEventsAttach() - attach an event buffer to a scope index
ScopeEventsDetach() - detach an event buffer from a scope index
ScopeCollectSignals() - collect a sample from each active signal
ScopeCollectionModeEnable() - enable periodic collection
ScopeCollectionModeDisable() - disable periodic collection
ScopeCollectionModeGet() - return the collection mode
ScopeChangeSampleRate() - change the sampling rate
ScopeSampleDivisorSet() - set the sample divisor for sub-sampling
ScopeRegisterSignalWithOffset() - register a signal with an offset
ScopeRegisterSignal() - register a signal
ScopeRemoveSignal() - remove a signal
ScopeRemoveMultipleSignals() - remove several similarly-named signals
ScopeActivateSignal() - activate a signal
ScopeActivateMultipleSignals() - activate multiple signals
ScopeDeactivateSignal() - deactivate a signal
ScopeDeactivateMultipleSignals() - deactivate a group of signals
ScopeInstallSignalWithOffset() - register and activate a signal with an offset
ScopeInstallSignal() - register and activate a signal
ScopeShowSignals() - print all registered signals
ScopeShowActiveSignals() - print all signal that are being collected
ScopeRegisterArray() - register an array of signals
ScopeInstallArray() - register and activate an array of signals
ScopeTriggerSet() - set the triggering parameters
ScopeTriggerGet() - return the current trigger parameters

DESCRIPTION This library provides a programmatic interface to Data Monitor real-time data collection and signal management, facilitating collection of time-histories of variables in your real-time program.

ScopeActivateMultipleSignals()

NAME **ScopeActivateMultipleSignals()** – activate multiple signals

SYNOPSIS

```
int ScopeActivateMultipleSignals
(
    const char *namePrefix,
    int
    scopeIndex
)
```

DESCRIPTION This function activates signals that have *namePrefix* as prefix. Activated signals can be selected for viewing from one of the many data-display windows (for example, **Plot**, **Monitor**, and so forth). **ScopeCollectSignals()** only collects samples of activated signals. A prefix of "." activates all signals.

RETURNS On success, the number of signals that have been activated.
On failure, it returns 0, indicating one of the following:

- the index *scopeIndex* is invalid or has not been initialized
- there is no signal with the *namePrefix* registered with index *scopeIndex*
- *namePrefix* is NULL or invalid

SEE ALSO **ScopeProbe()**, **ScopeInstallSignal()**, **ScopeInstallSignalWithOffset()**, **ScopeRegisterSignal()**, **ScopeRegisterSignalWithOffset()**, **ScopeActivateSignal()**, **ScopeDeactivateSignal()**, **ScopeRemoveSignal()**, **ScopeRemoveMultipleSignals()**

ScopeActivateSignal()

NAME **ScopeActivateSignal()** – activate a signal

SYNOPSIS

```
RTIBool ScopeActivateSignal
(
    const char *name,
    int scopeIndex
)
```

DESCRIPTION	This function activates a signal. Activated signals can be selected for viewing from one of the many data-display windows (for example, Plot , Monitor , and so forth). ScopeCollectSignals() only collects samples of activated signals.
RETURNS	On success, this function returns RTI_TRUE , indicating that the signal was activated. On failure, it returns RTI_FALSE , indicating one of the following: <ul style="list-style-type: none">▪ the index <i>scopeIndex</i> is invalid or has not been initialized▪ there is no signal named <i>name</i> registered with index <i>scopeIndex</i>
SEE ALSO	ScopeProbe() , ScopeInstallSignal() , ScopeInstallSignalWithOffset() , ScopeRegisterSignal() , ScopeRegisterSignalWithOffset() , ScopeActivateSignal() , ScopeDeactivateSignal() , ScopeRemoveSignal() , ScopeRemoveMultipleSignals()

ScopeChangeSampleRate()

NAME	ScopeChangeSampleRate() – change the sampling rate
SYNOPSIS	<pre>float ScopeChangeSampleRate (float newSampleRate, int scopeIndex)</pre>
DESCRIPTION	<p>This routine changes the amount of time that Data Monitor assumes passed between calls to ScopeCollectSignals(). It does not change the actual sampling rate, that is a responsibility of user code. Data Monitor uses the rate to calculate times between samples. If the rate is incorrect, these calculations will be in error.</p> <p>The parameter should be the frequency in samples per second of the calls to ScopeCollectSignals().</p>
RETURNS	On success, this function returns the old sampling rate. On failure, this function returns 0.0 indicating one of the following: <ul style="list-style-type: none">▪ <i>scopeIndex</i> is invalid or is not initialized▪ <i>rate</i> is invalid (≤ 0.0)
SEE ALSO	ScopeProbe() , ScopeCollectSignals() , ScopeInitServer()

ScopeCollectSignals()

NAME	ScopeCollectSignals() – collect a sample from each active signal
SYNOPSIS	<pre>void ScopeCollectSignals(int scopeIndex)</pre>
DESCRIPTION	<p>This routine should be called periodically to collect the values of signals. Data Monitor will assume that this function is called at the frequency set using ScopeChangeSampleRate(). However, this only affects the timing calculations made by Data Monitor.</p> <p>If a sample divisor is set and is greater than 1, then this routine will return without collecting data when a sample is to be skipped. For instance, with a sample divisor of 3, this routine will only actually collect the data every third time it is called. The other two times, it will simply return immediately.</p> <p>Furthermore, the behavior of this function depends on triggering. If the trigger is set and is waiting for the start condition to occur (ARMED), this function will return without collecting any data. Refer to ScopeTriggerSet() for more information.</p> <p>VxWorks users: the task that calls this routine should have floating point enabled (VX_FP_TASK set).</p> <p>Example code exists under the <i>src</i> directory.</p>
SEE ALSO	ScopeProbe() , ScopeChangeSampleRate() , ScopeCollectionModeEnable() , ScopeCollectionModeDisable() , ScopeTriggerSet()

ScopeCollectionModeDisable()

NAME	ScopeCollectionModeDisable() – disable periodic collection
SYNOPSIS	<pre>RTIBool ScopeCollectionModeDisable (int scopeIndex)</pre>
DESCRIPTION	<p>Use this function to disable periodic collection of active signals. Calling this function sets a flag that is examined by ScopeCollectSignals(). If the flag is set, ScopeCollectSignals() returns without sampling active signals. To turn collection on, use ScopeCollectionModeEnable().</p>
RETURNS	<p>On success, this function returns RTI_TRUE.</p> <p>On failure, this function returns RTI_FALSE if:</p>

- *scopeIndex* is invalid or is not initialized

SEE ALSO **ScopeProbe()**, **ScopeCollectSignals()**, **ScopeCollectionModeEnable()**,
ScopeCollectionModeGet()

ScopeCollectionModeEnable()

NAME **ScopeCollectionModeEnable()** – enables periodic collection.

SYNOPSIS `RTIBool ScopeCollectionModeEnable`
 (
 int scopeIndex
)

DESCRIPTION Use this function to re-enable periodic collection of active signals if it was turned off earlier using **ScopeCollectionModeDisable()**. By default (after calling **ScopeInitServer()**), the collection mode is enabled. Refer to **ScopeCollectionModeDisable()** for more information.

RETURNS On success, this function returns **RTI_TRUE**.
On failure, this function returns **RTI_FALSE** if:

- *scopeIndex* is invalid or is not initialized

SEE ALSO **ScopeProbe()**, **ScopeCollectSignals()**, **ScopeCollectionModeDisable()**,
ScopeCollectionModeGet()

ScopeCollectionModeGet()

NAME **ScopeCollectionModeGet()** – return the collection mode

SYNOPSIS `int ScopeCollectionModeGet`
 (
 int scopeIndex
)

DESCRIPTION Use this function to determine the current collection mode.

RETURNS On success, this function returns:

- **SCOPE_MODE_ENABLED** if collection mode is enabled

- `SCOPE_MODE_DISABLED` if collection mode is disabled

On failure, this function returns -1 if:

- *scopeIndex* is invalid or is not initialized

SEE ALSO

`ScopeProbe()`, `ScopeCollectSignals()`, `ScopeCollectionModeEnable()`,
`ScopeCollectionModeDisable()`

ScopeDeactivateMultipleSignals()

NAME

`ScopeDeactivateMultipleSignals()` – deactivate a group of signals

SYNOPSIS

```
int ScopeDeactivateMultipleSignals
(
    const char *namePrefix,
    int scopeIndex
)
```

DESCRIPTION

This function will remove all signals from the list of active signals for the selected *scopeIndex* which start with the specified *namePrefix*. Therefore, data will not be collected for these signals during `ScopeCollectSignals()`.

A prefix of "." deactivates all signals.

RETURNS

On success, the number of signals that have been deactivated.

On failure, this function returns 0, indicating one of the following:

- the index *scopeIndex* is invalid or has not been initialized
- there are no active signals with the *namePrefix* registered with index *scopeIndex*
- *namePrefix* is invalid

SEE ALSO

`ScopeProbe()`, `ScopeActivateMultipleSignals()`

ScopeDeactivateSignal()

NAME	ScopeDeactivateSignal() – deactivate a signal
SYNOPSIS	<pre>RTIBool ScopeDeactivateSignal (const char *name, int scopeIndex)</pre>
DESCRIPTION	This function will remove a signal from the list of activate signals. Therefore, data will not be collected for this signal during ScopeCollectSignals() .
RETURNS	On success, this function returns RTI_TRUE , indicating that the signal was deactivated. On failure, this function returns RTI_FALSE , indicating one of the following: <ul style="list-style-type: none">▪ the index <i>scopeIndex</i> is invalid or has not been initialized▪ there is no active signal named <i>name</i> registered with index <i>scopeIndex</i>
SEE ALSO	ScopeProbe() , ScopeActivateSignal()

ScopeEventsAttach()

NAME	ScopeEventsAttach() – attach an event buffer to a scope index
SYNOPSIS	<pre>int ScopeEventsAttach (int eventBufferSize, int scopeIndex)</pre>
DESCRIPTION	This function attaches an event buffer to a scope index. The eventsHandle returned by this function should be used to <i>throw</i> events. There is a maximum limit of four event buffers that can attach to an index. Throwing events into the same buffer from multiple threads is not recommended.
RETURNS	On success, this function returns a non-zero eventsHandle . On failure, it returns 0 if: <ul style="list-style-type: none">▪ <i>scopeIndex</i> is invalid or is not initialized▪ the maximum number of event buffers are already attached

SEE ALSO [ScopeProbe\(\)](#), [ScopeInitServer\(\)](#), [ScopeEventsDetach\(\)](#)

ScopeEventsCollect()

NAME [ScopeEventsCollect\(\)](#) – collect the value of a variable on the spot

SYNOPSIS

```
void ScopeEventsCollect
(
    int eventsHandle,
    int level,
    const char *eventId,
    void. ptrToVar, RTIAAtomicTypeId typeId
)
```

DESCRIPTION This function collects the value of one variable on the spot. The variable does not have to be static/global as required by [ScopeCollectSignals\(\)](#) and there is no need to perform registration/activation. This function is a low-overhead equivalent of [printf\(\)](#) for use in debugging real-time systems.

[ScopeEventsCollect\(\)](#) is actually a macro that calls the collection function [ScopeEventsCollectInternal\(\)](#) only if the verbosity level of this event is turned on. Thus, there is no overhead of a function call if the verbosity level of this event is turned off. The verbosity mask can be set using [ScopeEventsMaskSet\(\)](#).

NOTE: Calling this function from multiple threads with the same *eventsHandle* is not recommended. Therefore, use a separate *eventsHandle* (returned by [ScopeEventsAttach\(\)](#)) for each thread, which calls this function.

SEE ALSO [ScopeProbe\(\)](#), [ScopeEventsMessage\(\)](#), [ScopeEventsAttach\(\)](#), [ScopeEventsMaskSet\(\)](#)

ScopeEventsDetach()

NAME [ScopeEventsDetach\(\)](#) – detach an event buffer from a scope index

SYNOPSIS

```
RTIBool ScopeEventsDetach
(
    int eventsHandle
)
```

DESCRIPTION This function detaches an event buffer from an index.

RETURNS	On success, this function returns RTI_TRUE . On failure, it returns RTI_FALSE if: <ul style="list-style-type: none">▪ <i>eventsHandle</i> is invalid
SEE ALSO	ScopeProbe() , ScopeInitServer() , ScopeEventsAttach()

ScopeEventsMaskSet()

NAME	ScopeEventsMaskSet() – set the events verbosity mask
SYNOPSIS	<pre>RTIBool ScopeEventsMaskSet (int eventsMask, int scopeIndex)</pre>
DESCRIPTION	Data Monitor Events API allows for 32 levels of verbosity. This function sets the mask for the specified scope index. Each bit in mask corresponds to one verbosity level. Setting the mask to 0x00000001 turns off all levels except level 1 messages. The mask can be set anytime during the executing of the program and take effect immediately. By default, all levels are turned on for a scope index.
RETURNS	On success, this function returns RTI_TRUE . On failure, this function returns RTI_FALSE if: <ul style="list-style-type: none">▪ <i>scopeIndex</i> is invalid or is not initialized
SEE ALSO	ScopeProbe() , ScopeEventsCollect() , ScopeEventsMessage() , ScopeEventsAttach()

ScopeEventsMessage()

NAME	ScopeEventsMessage() – throw an event with the specified message
SYNOPSIS	<pre>void ScopeEventsMessage (int eventsHandle, int level, const char *message)</pre>

DESCRIPTION	<p>This function throws an event with the specified message string. This function is a low-overhead equivalent of printf() for use in debugging real-time systems.</p> <p>ScopeEventsMessage() is actually a macro that calls the collection function ScopeEventsMessageInternal() only if the verbosity level of this event is turned on. Thus, there is no overhead of a function call if the verbosity level of this event is turned off. The verbosity mask can be set using ScopeEventsMaskSet().</p> <hr/> <p>NOTE: Calling this function from multiple threads with the same <i>eventsHandle</i> is not recommended. Therefore, use a separate <i>eventsHandle</i> (returned by ScopeEventsAttach()) for each thread, which calls this function.</p> <hr/>
SEE ALSO	ScopeProbe() , ScopeEventsCollect() , ScopeEventsAttach() , ScopeEventsMaskSet()

ScopeInitServer()

NAME	ScopeInitServer() – initialize a <i>scope</i> index
SYNOPSIS	<pre>int ScopeInitServer (int sampleBufferSize, int signalBufferSize, int debugLevel, int scopeIndex)</pre>
DESCRIPTION	<p>Calls ScopeInitServerEx() with default priorities for scopeprobe and scopelink daemon threads. Refer to ScopeInitServerEx() for details.</p>
RETURNS	<p>On success, this function returns the initialized scope index.</p> <p>On failure, this function returns a negative number if:</p> <ul style="list-style-type: none"> ▪ <i>scopeIndex</i> is out of range ▪ all indices are occupied ▪ memory allocation failed: it failed to create daemon threads (which will happen if the threads failed to bind to TCP ports, or if the priorities specified are not valid on the target operating system)
SEE ALSO	ScopeProbe() , ScopeInitServerEx()

ScopeInitServerEx()

NAME `ScopeInitServerEx()` – initialize a scope index

SYNOPSIS

```
int ScopeInitServerEx
(
    int sampleBufferSize,
    int signalBufferSize,
    int debugLevel,
    int probeDaemonPriority,
    int linkDaemonPriority,
    int scopeIndex
)
```

DESCRIPTION Initializes the scope index *scopeIndex*. If *scopeIndex* is -1, then this function initializes the next uninitialized scope index. This should be called once for each index (on VxWorks, normally at boot time). Calling it multiple times for the same index is harmless though.

The *sampleBufferSize* parameter specifies the size of the target data buffer in bytes. This buffer is used to store the data samples for the active signals. Data samples for all types other than **double** are saved as 4-byte values. Data samples for doubles are saved as 8-byte values. Thus, the maximum number of samples that can fit in the buffer will range between $((\text{sampleBufferSize} / 8) - 1) / \text{numberOfSignals}$ and $((\text{sampleBufferSize} / 4) - 1) / \text{numberOfSignals}$, rounded down to the nearest integer.

If *sampleBufferSize* ≤ 0 , it defaults to (32*1024). Otherwise, if it is less than 1024, a value of 1024 is used instead.

The *signalBufSize* parameter specifies the size of the target signal buffer in bytes. This buffer is used to store the information specific to each signal (such as the name, units and type). The space taken by a signal gets reclaimed (for registering other signals) when the signal is removed. The information stored for a signal takes up 28 bytes plus the number of bytes (including the terminating null character) it takes to store the signal name and units. Note that a signal registered twice, under different scope indices, counts as two registered signals.

If *signalBufferSize* ≤ 0 , it defaults to (32*1024). If *signalBufferSize* > 0 and < 1024 it defaults to 1024.

The *scopeprobeDaemonPriority* and *scopelinkDaemonPriority* parameters specify the scheduling priority levels for the **scopeprobe** and **scopelink** daemon threads that are spawned by this function. Refer to the documentation on threads and scheduling for the operating system that you are using, for valid priority levels. If you do not want to specify a priority level, you can pass a nonpositive value for this parameter. In that case, suitable priority levels will automatically be chosen.

The *scopeIndex* parameter should be an integer within the range of 0 to 127. If the value "-1" is passed, then an index will be assigned automatically and returned.

Normally, for each scope index, a target spawns real-time daemons (**scopeprobe** and **scopelink**) which communicate to the host over the network, using TCP/IP. If you are using Tornado, and wish to use the Tornado WTX protocol instead, you must load the Data Monitor WTX target library (**libscopewtx.so**). In that case, the daemons will use the WTX protocol to communicate even if IP is available. Note that the protocol is called WTX, even though the target is talking to the WDB daemon.

RETURNS	On success, this function returns the initialized scope index. On failure, this function returns a negative number if: <ul style="list-style-type: none"> ▪ <code>scopeIndex</code> is out of range ▪ all indices are occupied ▪ memory allocation failed: it failed to create daemon threads (which will happen if the threads failed to bind to TCP ports, or if the priorities specified are not valid on the target operating system).
SEE ALSO	ScopeProbe() , ScopeShutdown() , ScopeEventsAttach() , ScopeEventsDetach()

ScopeInstallArray()

NAME	ScopeInstallArray() – register and activate an array of signals
SYNOPSIS	<pre>int ScopeInstallArray (const char *name, int elementsInArray, * .units, void .ptrToStaticArray, const char *type, int scopeIndex)</pre>
DESCRIPTION	This function calls ScopeRegisterArray() followed by ScopeActivateMultipleSignals() . It exists purely for convenience. Users should refer to ScopeRegisterArray() and ScopeActivateMultipleSignals() for details.
EXAMPLES	<pre>float global_array[100]; int main() { static double static_array[50]; // Initialize ScopeIndex here. See ScopeInitServer().</pre>

```
ScopeInstallArray("my_global_array1", 100, "none", &global_array,  
                  "float", ScopeIndex);  
ScopeInstallArray("my_static_array1", 50, "none", &static_array,  
                  "double", ScopeIndex);  
  
while ( ) {  
    ScopeCollectSignals( );  
}  
// Shutdown ScopeIndex. See ScopeShutdown( ).  
}
```

RETURNS On success, this function returns the numbers of array elements successfully installed.

On failure, it returns zero, indicating one of the following:

- the index (*scopeIndex*) is invalid or has not been initialized
- the type specified (*type*) is not recognized
- the parameters passed are invalid
- there is no more space in the signal buffer

SEE ALSO [ScopeProbe\(\)](#), [ScopeRegisterArray\(\)](#), [ScopeActivateMultipleSignals\(\)](#),
[ScopeDeactivateMultipleSignals\(\)](#), [ScopeInstallSignal\(\)](#)

ScopeInstallSignal()

NAME [ScopeInstallSignal\(\)](#) – register and activate a signal

SYNOPSIS

```
RTIBool ScopeInstallSignal  
(  
    const char *name,  
    const char *units,  
    void .ptrToStaticVar  
    const char *type,  
    int scopeIndex  
)
```

PARAMETERS

name
A unique name to identify the signal.

units
User specified unit of measurement, for identification.

ptrToStaticVar
Must point to a static (or global) storage location.

type
Must be one of the accepted types such as: *double*, *float*, *int*, and so forth. See [ScopeProbe\(\)](#) for details on types.

DESCRIPTION This function calls **ScopeRegisterSignal()** followed by **ScopeActivateSignal()**. It exists purely for backwards compatibility and convenience. Users should refer to **ScopeRegisterSignal()** and **ScopeActivateSignal()** for details.

EXAMPLES

```
float global_var;
float .global_ptr;

int main( )
{

    static double static_var;
    static double .static_ptr;

    // Initialize ScopeIndex here. See ScopeInitServer( ).

    ScopeInstallSignal("my_global_var1", "none", &global_var,
                      "float", ScopeIndex);
    ScopeInstallSignal("my_static_var1", "none", &static_var,
                      "double", ScopeIndex);

    global_ptr = (float .) calloc(1, sizeof(.global_ptr));
    static_ptr = (double .) calloc(1, sizeof(.static_ptr));
    ScopeInstallSignal("my_global_var2", "none", global_ptr,
                      "double", ScopeIndex);
    ScopeInstallSignal("my_static_var2", "none", static_ptr,
                      "double", ScopeIndex);

    while ( ) {
        ScopeCollectSignals( );
    }
    // Shutdown ScopeIndex. See ScopeShutdown( ).
}
```

RETURNS On success, this function returns **RTI_TRUE**, indicating that the signal was installed.

On failure, it returns **RTI_FALSE**, indicating one of the following:

- the index (*scopeIndex*) is invalid or has not been initialized
- the type specified (*type*) is not recognized
- the parameters passed are invalid
- there is no more space in the signal buffer

SEE ALSO **ScopeProbe()**, **ScopeRegisterSignal()**, **ScopeActivateSignal()**, **ScopeDeactivateSignal()**, **ScopeRemoveSignal()**, **ScopeRemoveMultipleSignals()**

ScopeInstallSignalWithOffset()

NAME	ScopeInstallSignalWithOffset() – register and activate a signal with an offset
SYNOPSIS	<pre>RTIBool ScopeInstallSignalWithOffset (const char *name, const char *units, void .ptrToStaticVar, const char *type, int offset, int scopeIndex)</pre>
DESCRIPTION	Registers and activates a signal with an offset.
PARAMETERS	<p><i>name</i> A unique name to identify the signal.</p> <p><i>units</i> User specified unit of measurement, for identification.</p> <p><i>ptrToStaticVar</i> Must point to a static (or global) storage location.</p> <p><i>type</i> Refers to the type of the variable at the given <i>offset</i>.</p> <p><i>offset</i> Number of bytes from <i>ptrToStaticVar</i> to collect data from.</p>
DESCRIPTION	This function calls ScopeRegisterSignalWithOffset() followed by ScopeActivateSignal() . It exists purely for convenience. Users should refer to ScopeRegisterSignalWithOffset() and ScopeActivateSignal() for details.
EXAMPLES	<p>The following illustrates how to use ScopeInstallSignalWithOffset()</p> <pre>typedef struct ArmData_s { float PosX; float PosY; unsigned char Type; double .Vel; // Just to demonstrate pointers and offsets. } ArmData_t; int main () {</pre>

```
ArmData_t .LeftArmData = (ArmData_t .)calloc(1,
                                             sizeof( ));
LeftArmData->Vel = (double .)calloc(1, sizeof( ));
// Initialize ScopeIndex here. See ScopeInitServer( ).
// Notice that you can pass just the address of LeftArmData,
// not the individual fields.
// The type "float" refers to the type of LeftArmData->PosX.
ScopeInstallSignalWithOffset("LeftArm/PosX", "meters",
                             &LeftArmData, "float",
                             offsetof(ArmData_t, PosX), 0);
ScopeInstallSignalWithOffset("LeftArm/PosY", "meters",
                             &LeftArmData, "float",
                             offsetof(ArmData_t, PosY), 0);
ScopeInstallSignalWithOffset("LeftArm/Type", "meters",
                             &LeftArmData, "uchar",
                             offsetof(ArmData_t, Type), 0);

// The type "double ." refers to the type of the member Vel.
ScopeInstallSignalWithOffset("LeftArm/Vel", "meters",
                             &LeftArmData, "double .",
                             offsetof(ArmData_t, Vel), 0);

while ( ) {
    // Set the data in the member LeftArmData.
    ScopeCollectSignals( );
}
// Shutdown ScopeIndex. See ScopeShutdown( ).
}
```

RETURNS On success, this function returns **RTI_TRUE**, indicating that the signal was installed.

On failure, it returns **RTI_FALSE**, indicating one of the following:

- the index (*scopeIndex*) is invalid or has not been initialized
- the type specified (*type*) is not recognized
- the parameters passed are invalid
- there is no more space in the signal buffer

SEE ALSO **ScopeProbe()**, **ScopeRegisterSignal()**, **ScopeActivateSignal()**, **ScopeDeactivateSignal()**, **ScopeRemoveSignal()**, **ScopeRemoveMultipleSignals()**

ScopePrintVersion()

NAME **ScopePrintVersion()** – print the version number of the Data Monitor target library

SYNOPSIS `void ScopePrintVersion(void)`

DESCRIPTION Prints the version number of the Data Monitor target library.

SEE ALSO [ScopeProbe\(\)](#), [ScopeProbe\(\)](#)

ScopeRegisterArray()

NAME [ScopeRegisterArray\(\)](#) – register an array of signals

SYNOPSIS

```
int ScopeRegisterArray
(
    const char *name,
    int elementsInArray,
    const char *units,
    void .ptrToStaticArray,
    const char *type,
    int scopeIndex
)
```

DESCRIPTION This function registers an array (one dimensional) of signals by calling **ScopeRegisterSignal()** for each element in the array. An array modifier "[*i*]" is appended to the signal name followed by the index of the element.

A valid signal name should not contain asterisks or blank spaces. Invalid characters in signal name (*name*) would be automatically replaced with underscores.

It is the responsibility of the caller to ensure that the address passed as *ptrToStaticArray* points to a valid memory location. In particular, a bad address may cause a **Bus Error** or **Segmentation Fault** to occur within **ScopeCollectSignals()**.

EXAMPLES

```
float global_array[100];

int main( )
{
    static double static_array[50];

    // Initialize ScopeIndex here. See ScopeInitServer( ).
    ScopeRegisterArray("my_global_array1", 100, "none",
        &global_array,
        "float", ScopeIndex);
    ScopeRegisterArray("my_static_array1", 50, "none",
        &static_array, "double", ScopeIndex);

    ScopeActivateMultipleSignals("my_global_array1", ScopeIndex);
    ScopeActivateMultipleSignals("my_static_array1", ScopeIndex);

    while ( ) {
        ScopeCollectSignals( );
    }

    // Shutdown ScopeIndex. See ScopeShutdown( ).
}
```

RETURNS	On success, this function returns the numbers of array elements successfully registered. On failure, it returns zero, indicating one of the following: <ul style="list-style-type: none">▪ the index <i>scopeIndex</i> is invalid or has not been initialized▪ the type specified <i>type</i> is not recognized▪ the parameters passed are invalid▪ there is no more space in the signal buffer
SEE ALSO	ScopeProbe() , ScopeRegisterSignal() , ScopeInstallArray() , ScopeActivateMultipleSignals() , ScopeDeactivateMultipleSignals()

ScopeRegisterSignal()

NAME **ScopeRegisterSignal()** – register a signal

SYNOPSIS

```
RTIBool ScopeRegisterSignal
(
    const char *name,
    const char *units,
    void .ptrToStaticVar
    const char *type,
    int scopeIndex
)
```

DESCRIPTION This function registers a *signal*. A registered signal is one that is made known to the Data Monitor signal manager, but not to any other window such as the **Plot** window. It has the possibility of becoming *activated* through the signal manager or through the target function **ScopeActivateSignal()**. Activated signals can be selected for display from one of the many data-display windows (for instance, **Plot**, **Monitor**, and so forth).

A signal is any variable in the program, with the caveat that it must have a valid value at the instant that **ScopeCollectSignals()** is called. Thus, most anything can be installed as a signal, with the exception of automatic stack variables whose scope does not include the **ScopeCollectSignals()** call.

A valid signal name should not contain asterisks or blank spaces. Invalid characters in signal *name* would be automatically replaced with underscores.

It is the responsibility of the caller to ensure that the address passed *ptrToStaticVar* points to a valid memory location. In particular, a bad address may cause a **Bus Error** or **Segmentation Fault** to occur within **ScopeCollectSignals()**.

EXAMPLES

```
float global_var;
float .global_ptr;
```

```
int main( )
{
    static double static_var;
    static double .static_ptr;

    // Initialize ScopeIndex here. See ScopeInitServer( ).

    ScopeRegisterSignal("my_global_var1", "none", &global_var,
                        "float", ScopeIndex);
    ScopeRegisterSignal("my_static_var1", "none", &static_var,
                        "double", ScopeIndex);

    global_ptr = (float .) calloc(1, sizeof(.global_ptr));
    static_ptr = (double .) calloc(1, sizeof(.static_ptr));

    ScopeRegisterSignal("my_global_var2", "none", global_ptr,
                        "double", ScopeIndex);
    ScopeRegisterSignal("my_static_var2", "none", static_ptr,
                        "double", ScopeIndex);

    // Activate the registered signals here. See ScopeActivateSignal( ).
    while ( ) {
        ScopeCollectSignals( );
    }

    // Shutdown ScopeIndex. See ScopeShutdown( ).
}
```

RETURNS On success, this function returns **RTI_TRUE**, indicating that the signal was registered.
On failure, it returns **RTI_FALSE**, indicating one of the following:

- the index *scopeIndex* is invalid or has not been initialized
- the type specified *type* is not recognized
- the parameters passed are invalid
- there is no more space in the signal buffer

SEE ALSO **ScopeProbe(), ScopeActivateSignal(), ScopeDeactivateSignal(), ScopeInstallSignal(), ScopeRegisterSignalWithOffset()**

ScopeRegisterSignalWithOffset()

NAME **ScopeRegisterSignalWithOffset()** – register a signal with an offset

SYNOPSIS

```
RTIBool ScopeRegisterSignalWithOffset
(
    const char *name,
    const char *units,
    void .ptrToStaticVar,
    const char *type,
    int offset,
    int scopeIndex
)
```

DESCRIPTION This function has the same functionality as **ScopeRegisterSignal()**, except callers can specify an offset from *ptrToStaticVar*. This offset refers to a variable in a structure that a caller wants to sample. However, the address of the variable may not be known at register time, hence the offset.

A valid signal name should not contain asterisks or blank spaces. Invalid characters in signal *name* would be automatically replaced with underscores.

It is the responsibility of the caller to ensure that the address and offset parameters point to a valid memory location. In particular, a bad address may cause a **Bus Error** or **Segmentation Fault** during **ScopeCollectSignals()**.

Ideally, this function would be used to sample variables in an array of structures. The caller would register a structure pointer with an offset to a variable they want sampled. Therefore, by changing the pointer callers can sample the same variable at different locations in the array, see below for examples.

EXAMPLES

```
// An example structure.
typedef struct TestData_s {

    float field1;
    double field2;
} TestData_t;

// Here is an array that is read in.
#define MAXLENGTH 100
TestData_t TestDataArray[MAXLENGTH];
TestData_t .TestDataPtr = &TestDataArray[0];

int main( )
{
    int i = 0;
```

```
// Initialize ScopeIndex here. See ScopeInitServer( ).
// Notice we are using the new hierarchical naming feature.
ScopeRegisterSignalWithOffset("test/field1", "volts",
                              &TestDataPtr, "float",
                              offsetof(TestData_t, field1),
                              ScopeIndex);
ScopeRegisterSignalWithOffset("test/field2", "volts",
                              &TestDataPtr, "double",
                              offsetof(TestData_t, field2),
                              ScopeIndex);

// Activate the registered signals here.
// See ScopeActivateSignal( ).

for (i = 0; i < MAXLENGTH; i++) {
    // Read in TestDataArray[i].
    // Set TestDataPtr to the new data.
    TestDataPtr = &TestDataArray[i];
    ScopeCollectSignals( );
}

// Shutdown ScopeIndex. See ScopeShutdown( ).
}
```

RETURNS On success, this function returns **RTL_TRUE**, indicating that the signal was registered.

On failure, it returns **RTL_FALSE**, indicating one of the following:

- the index *scopeIndex* is invalid or has not been initialized
- the type specified *type* is not recognized
- the parameters passed are invalid
- there is no more space in the signal buffer

SEE ALSO [ScopeProbe\(\)](#), [ScopeRegisterSignal\(\)](#), [ScopeActivateSignal\(\)](#), [ScopeInstallSignal\(\)](#), [ScopeInstallSignalWithOffset\(\)](#), [ScopeRemoveSignal\(\)](#), [ScopeRemoveMultipleSignals\(\)](#), [ScopeDeactivateSignal\(\)](#)

ScopeRemoveMultipleSignals()

NAME [ScopeRemoveMultipleSignals\(\)](#) – remove several similarly-named signals

SYNOPSIS

```
int ScopeRemoveMultipleSignals
(
    const char *namePrefix,
    int scopeIndex
)
```


DESCRIPTION	This function removes a set of installed signals that have <i>namePrefix</i> as prefix. If <i>namePrefix</i> is "", all signals will be removed.
RETURNS	On success, returns the number of signals removed. On failure, this function returns 0, indicating one of the following: <ul style="list-style-type: none">▪ <i>namePrefix</i> is NULL▪ there is no signal named with prefix <i>namePrefix</i> registered with index <i>scopeIndex</i>▪ the index <i>scopeIndex</i> is out of range or if index is not initialized
SEE ALSO	ScopeProbe(), ScopeRemoveSignal()

ScopeRemoveSignal()

NAME	ScopeRemoveSignal() – remove a signal
SYNOPSIS	<pre>RTIBool ScopeRemoveSignal (const char *name, int scopeIndex)</pre>
DESCRIPTION	This function deactivates and unregisters a signal. This will invalidate any partially filled collection buffer. After this call users can not activate this signal.
RETURNS	On success, this function returns RTI_TRUE , indicating that the installed signal was removed. On failure, it returns RTI_FALSE , indicating one of the following: <ul style="list-style-type: none">▪ the index <i>scopeIndex</i> is invalid or has not been initialized▪ there is no signal named <i>name</i> registered with index <i>scopeIndex</i>
SEE ALSO	ScopeProbe(), ScopeInstallSignal(), ScopeInstallSignalWithOffset(), ScopeRegisterSignal()

ScopeSampleDivisorSet()

NAME	ScopeSampleDivisorSet() – set the sample divisor for sub-sampling
SYNOPSIS	<pre>int ScopeSampleDivisorSet (int newSampleDivisor, int scopeIndex)</pre>
DESCRIPTION	This function sets the sample divisor for this index. From then on, ScopeCollectSignals() will succeed only once in <i>newSampleDivisor</i> number of times it is invoked.
RETURNS	On success, this function returns the old sample divisor. On failure, this function returns 0 indicating one of the following: <ul style="list-style-type: none">▪ <i>scopeIndex</i> is invalid or is not initialized▪ <i>newSampleDivisor</i> is invalid (Range: 1 - 10,000)
SEE ALSO	ScopeProbe() , ScopeCollectSignals() , ScopeInitServer()

ScopeShowActiveSignals()

NAME	ScopeShowActiveSignals() – print all signals that are being collected
SYNOPSIS	<pre>void ScopeShowActiveSignals(int scopeIndex)</pre>
DESCRIPTION	This function prints a formatted list of all currently installed signals to the standard output. The current signal value is also printed.
SEE ALSO	ScopeProbe() , ScopeInstallSignal() , ScopeInstallSignalWithOffset() , ScopeRegisterSignal() , ScopeRegisterSignalWithOffset()

ScopeShowSignals()

NAME	ScopeShowSignals() – print all registered signals.
SYNOPSIS	<pre>void ScopeShowSignals (int scopeIndex)</pre>
DESCRIPTION	This function prints a formatted list of all currently registered signals to the standard output. The current signal value is also printed.
SEE ALSO	ScopeProbe() , ScopeInstallSignal() , ScopeInstallSignalWithOffset() , ScopeRegisterSignal() , ScopeRegisterSignalWithOffset()

ScopeShutdown()

NAME	ScopeShutdown() – shut down a scope index
SYNOPSIS	<pre>RTIBool ScopeShutdown (int scopeIndex)</pre>
DESCRIPTION	This function shuts down the scope index <i>scopeIndex</i> . Event buffers, if any, associated with this index are also shutdown.
RETURNS	On success, this function returns RTI_TRUE . On failure, this function returns RTI_FALSE if: <ul style="list-style-type: none">▪ <i>scopeIndex</i> is invalid or not initialized
SEE ALSO	ScopeProbe() , ScopeInitServer() , ScopeInitServerEx() , ScopeEventsAttach() , ScopeEventsDetach()

ScopeTriggerGet()

NAME	ScopeTriggerGet() – return the current trigger parameters
SYNOPSIS	<pre>RTIBool ScopeTriggerGet (int .pTriggerMode, int .pTriggerRearm, ScopeTriggerInfoPtr pTriggerStart, ScopeTriggerInfoPtr pTriggerStop, int scopeIndex)</pre>
PARAMETERS	<p><i>pTriggerRearm</i> Gets the rearm flag: 0 (Rearm flag is not set), 1 (Rearm flag is set)</p> <p><i>pTriggerStart</i> Gets the start condition structure.</p> <p><i>pTriggerStop</i> Gets the stop condition structure.</p>
DESCRIPTION	<p>This function returns the current trigger parameters.</p> <p>If pTriggerMode returns:</p> <ul style="list-style-type: none">▪ SCOPE_TRIG_MODE_DISABLED, then there is no trigger set for that index.▪ SCOPE_TRIG_MODE_ARMED, there is a trigger set for that index and is waiting for the start condition to occur. This also means that no data is being collected by ScopeCollectSignals().▪ SCOPE_TRIG_MODE_TRIGGERING, there is a trigger set for that index and is waiting for the stop condition to occur.
RETURNS	<p>On success, this function returns RTI_TRUE.</p> <p>On failure, this function returns RTI_FALSE if:</p> <ul style="list-style-type: none">▪ <i>scopeIndex</i> is invalid or has not been initialized.
SEE ALSO	ScopeProbe(), ScopeTriggerSet(), ScopeCollectSignals()

ScopeTriggerSet()

NAME **ScopeTriggerSet()** – set the triggering parameters

SYNOPSIS

```
RTIBool ScopeTriggerSet
(
    ScopeTriggerInfoPtr pTriggerStart,
    ScopeTriggerInfoPtr pTriggerStop,
    RTIBool triggerRearm,
    int scopeIndex
)
```

DESCRIPTION This function sets up a trigger. Triggering is a way to control when and for how long periodic data (collected using **ScopeCollectSignals()**) is collected. A trigger consists of a start condition and a stop condition. After the trigger is armed using **ScopeTriggerSet()**, all calls to **ScopeCollectSignals()** will return without collecting data. Data collection will resume only after the specified start condition is met. The start condition can be based on a signal or an event or a trigger can be set with no start condition (trigger immediately option). The stop condition can be based on a signal or an event. Stop condition for a trigger based on a set time period is available only from the Data Monitor GUI.

The trigger will expire as soon as the stop condition is met. Data collection will continue as before after the trigger expires. If the **triggerRearm** flag is set, the trigger will again be set with the values passed.

To set a trigger, fill out the following fields in the **ScopeTriggerInfo** structure:

source

The source of the condition. Can be one of:

- **SCOPE_TRIG_SRC_SIGNAL** (triggers on a signal)
- **SCOPE_TRIG_SRC_EVENT** (triggers on an event)
- **SCOPE_TRIG_SRC_NOW** (triggers immediately)

slope

The slope of the signal as it passes the level in order to trigger:

- **SCOPE_TRIG_SLOPE_POS** (positive slope)
- **SCOPE_TRIG_SLOPE_NEG** (negative slope)
- **SCOPE_TRIG_SLOPE_ANY** (any slope)

This parameter is relevant only if the source is a signal.

level

A value that the signal must reach in order to trigger. This parameter is relevant only if the source is a signal.

signal

The name of the signal to trigger on.

eventId

The event to trigger on.

To set a trigger based on a *signal*, fill out *source*, *slope*, *level* and *signal*. To set a trigger based on an *event*, fill out the *eventId*. To set the trigger to start immediately, fill out *source* (set it to **SCOPE_TRIG_SRC_NOW**).

For all the trigger modes listed above, pass in a non-NULL value to parameters that are not used in that mode.

To disable the trigger, pass NULL for **pTriggerStart** and **pTriggerStop**.

RETURNS

On success, this function returns **RTI_TRUE**, indicating that the trigger was set.

On failure, this function returns **RTI_FALSE** if:

- *scopeIndex* is invalid or is not initialized.

SEE ALSO

ScopeProbe(), **ScopeTriggerGet()**, **ScopeCollectSignals()**

C

Data Monitor Demo Program

- [C.1 Introduction 323](#)
- [C.2 Source Code for VxWorks 324](#)
- [C.3 Source Code for Linux 334](#)

C.1 Introduction

This appendix contains a listing of the demonstration source code file(s) provided with the Wind River Data Monitor distribution. This includes example source code for an application program that installs signals to Data Monitor.

The demo program is an example of a simple application that uses Data Monitor. The example application plots a number of sinusoids and contains a simple control system with a simulated physical system that is noisy. It is meant only to illustrate some of the Data Monitor features.

C.2 Source Code for VxWorks

The file and its makefile are located at:

```
WIND_SCOPETOOLS_BASE\target\src\scopedemo
```

where `WIND_BASE` (an environment variable of the same name) is the root of the tree where you installed the Run-Time Analysis Tools. The `scopedir\src\target\src\scopedemo` directory contains the `vxdemo.c` demo code file (see below), and a makefile (see [Makefile for vxdemo.c](#), p.332), to help you recompile the demonstration program.



NOTE: You must edit the **makefile** to have it reflect the file structure of your system.

VxWorks vxdemo.c Program

```
\- A demo for Data Monitor. */

/*
modification history
-----
5.4a,25sep00,ss Modified ln. 165 to properly support random numbers
in VxWorks versions < 5.1
5.4a,18sep00,gah added support for protection domain testing
5.4a,17jun00,vwc Better cleanup of memory by doing semDelete().
Also use flag to cause main task to terminate rather
than taskDelete()
5.4a,12jun00,vwc Add hook routine to be run in sample loop. Install
Scope's debug signals by default.
5.3c,30mar00,vwc Added param to ScopeDemo to set data-buf sz
5.3a,09jun99,laf Changed buffer sizes - we have 17 variables, not 16,
don't need quite as much memory for either buffer.
If ScopeInitServer fails (usually due to lack of
target memory), don't spawn the other tasks.
5.1f,17may99,laf Double data buffer size
5.1e,15apr99,sda Changed ordering of statements in Shutdown to safer.
5.1e,02apr99,laf Provide Shutdown functionality to clean up.
5.1a,26oct98,nm ScopeInitServer() now takes two buffer sizes.
5.1a,16oct98,nm Set WtxOverride to 1, not to true.
5.1a,12aug98,nm Updated for scope 5.1 release.
RTI,20dec92,sas Added ScopeIndex. Removed support for VxWorks 4.x
RTI,25nov92,sas Ported to VxWorks 5.1
RTI,06may92,sas Added rebootHookAdd to insure reboot success.
Made VxWorks 5.0 the default.
```



```
RTI,03apr92,sas  Converted to mangan format.
RTI,07nov91,sas  Added plant simulation.
RTI,01sep89,sas  written.
*/

/*

DESCRIPTION:

    This file contains code to start a simple synchronous sampler, and
    install and de-install a few demo signals.  It also contains a simple
    double-integrator plant simulator.

    To run the demo (this assumes you have already loaded Data Monitor):
rlogin target
cd "scopedir/lib/m68kVx5.1"
ld 1 < scopedemo.1o
ScopeDemo

    To recompile:
cc68k -I/local/VxWorks/h -Iscopedir/include -c scopedemo.c

    Useful things to play with:
Omega = (float) 80;
ScopeRemoveMultipleSignals("sin");
InstallFakeSignals(40)
ScopeDemoSetRate(20.0)

    A makefile is provided; it can be used to compile this code.  You
    should first edit it to reflect your system's file structure.

    This is a complete VxWorks application.  To utilize Data Monitor in
    your system, all you need is a call to ScopeInitServer in your
    startup script, a call to ScopeCollectSignals somewhere in your
    regular processing cycle, and a few calls to ScopeInstallSignal.

WARNING:

    With 64 signals installed, the calculation of the sin functions below
    (FakeSignals()) can only be done at about 200 Hz (on a 16MHz 68020).
    Thus, setting the sample rate higher than this will starve both tasks
    tScopeProbeDaemon and tScopeLinkDaemon; Scope will not be able to
    display any data.
    This is intentional---Scope always strives for minimal impact on the
    real-time system.

NOTES:

    This code normally works using taskDelay.  It can also work by
    attaching a semaphore (sampleSemaphore) to the Aux clock interrupt,
    the recommended means of executing periodic functions in VxWorks.  If
    your processor does have an aux-clock, set the "useAuxClock"
    parameter to 1.
```

```
SEE ALSO:
    ScopeProbe(2), scope(1), scopedemo2(3)
----- */
/* $Id: scopedemo.c,v 1.1 2003/03/27 17:52:12 Exp $ */

/* (c) Copyright Real-Time Innovations, Inc., 1999. All rights reserved. */

#include <stdio.h> /* Change to stdioLib.h for 5.0 */
#include <math.h>

#include "vxWorks.h"
#include "taskLib.h"
#include "sysLib.h"
#include "semLib.h"
#include "rtilib/vxVersions.h"
#include "rtilib/rti_types.h"
#include "rtilib/rti_endian.h"
#include "scope/scope.h"

#define MAXPHASES(128)
#define PHASESHIFT(0.1)
#define SQUARECOUNT(20)

float square= 1.0;
struct ExampleStruct {
    float sint;
    float sin2t;
    float sin3t;
    float cost;
} *singroup;
float phases[MAXPHASES];
float Omega = 1.0;

int NumberOfPhases = 0;

float Pos = 0.0;
float Vel = 0.0;
float Acc = 0.0;

float Posdes = 1.0;
float Veldes = 0.0;
float Kp = 10;
float Kv = 4;

float NoiseMag = 0.1;

float Time = 0.0;
float Dt = 0.005;

SEM_ID SampleSemaphore = NULL;
int ScopeIndex = -1;
int ScopeHandle = -1;
```

```
int tid = 0;
static int ScopeDemoShutdownRequest = 0;
static int ScopeNoServer = 0;

void (*SampleHookFunc)(void *) = NULL;
void *SampleHookParam = NULL;

void SampleHookSet( void (*func)(void *), void *param )
{
    SampleHookFunc = func;
    SampleHookParam = param;
}

static
/* Calculate a set of simple wave signals, paying absolutely no attention to
   efficiency. */
void SineWaves(float t)
{
    register int i;
    static int squareCount = SQUARECOUNT;
    register float wt = t * Omega;

    float sint, cost;

    if(--squareCount <= 0) {
        squareCount = SQUARECOUNT;
        square = 1.0 - square;
    }
    singroup->sint = sin(wt);
    singroup->cost = cos(wt);
    singroup->sin2t = sin(2 * wt);
    singroup->sin3t = sin(3 * wt);
    for(i=0; i<NumberOfPhases; i++) {
        phases[i] = sin(wt + PHASESHIFT*i);
    }

    sint = singroup->sint;
    cost = singroup->cost;

    /* Throw an event for when Sin and Cos are calculated. */
    if((singroup->sint > 0.1) && (singroup->sint <= 0.2)) {
        ScopeEventsCollect(ScopeHandle, 1, "Sine-0.1-0.2", (void *) &sint,
                           RTI_FLOAT32ID);
    }

    if((singroup->cost > 0.5) && (singroup->cost <= 0.6)) {
        ScopeEventsCollect(ScopeHandle, 1, "Cosine-0.5-0.6", (void *) &cost,
                           RTI_FLOAT32ID);
    }
}
```

```
static
/* Return a uniform random variable between a and b. */
float Noise(float a, float b)
{
    float result;

    result = (((float) rand() / RAND_MAX) - 0.5);
    return (result * (b - a) + (a + b) / 2.);
}

static
/* This "plant" is an Euler double integrator. */
void Plant(float t)
{
    Vel += Acc * Dt;
    Vel += Noise(-NoiseMag, NoiseMag);
    Pos += Vel * Dt;
}

static
void Control(float t)
{
    static int stepCount = 300;
    if(stepCount-- <= 0) {
        stepCount = 300;
        Posdes = -Posdes;

        /* Throw an event whenever position desired changes. */
        ScopeEventsCollect(ScopeHandle, 2, "PosChangeEvent", (void *) &Posdes,
                           RTI_FLOAT32ID);
    }

    Acc = Kp*(Posdes - Pos) + Kv*(Veldes - Vel);
}

static
void Sample(float t, int noCollection)
{
    SineWaves(t);
    Plant(t);
    Control(t);
    if (noCollection == 0) { ScopeCollectSignals(ScopeIndex); }
    if (SampleHookFunc != NULL) {
        (*SampleHookFunc)(SampleHookParam);
    }
}

static
void InstallFakeSignals(int num)
{
    register int i;
    char str[132];

    if(num > MAXPHASES) {num = MAXPHASES;}
    NumberOfPhases = num;
}
```

```

/* Install signals for debugging */

ScopeInstallSignal("Square", "volts", &square, "float", ScopeIndex);
singroup = (struct ExampleStruct *) calloc(1, sizeof(struct
ExampleStruct));
singroup->sint = 0.0;
singroup->sin2t = 0.0;
singroup->sin3t = 0.0;
singroup->cost = 1.0;
ScopeInstallSignal("Sine", "volts", &(singroup->sint), "float",
ScopeIndex);
ScopeInstallSignal("Cosine", "volts", &(singroup->cost), "float",
ScopeIndex);
ScopeInstallSignal("Sine2T", "volts", &(singroup->sin2t), "float",
ScopeIndex);
ScopeInstallSignal("Sine3T", "volts", &(singroup->sin3t), "float",
ScopeIndex);

ScopeInstallSignal("Pos", "meters", &Pos, "float", ScopeIndex);
ScopeInstallSignal("PosDesired", "meters", &Posdes, "float", ScopeIndex);
ScopeInstallSignal("Vel", "m/s", &Vel, "float", ScopeIndex);
ScopeInstallSignal("Acc", "m/s/s", &Acc, "float", ScopeIndex);

for(i=0; i<num; i++) {
    sprintf(str, "sin/sin(t+%.1f)", PHASESHIFT*i);
    ScopeInstallSignal(str, "volts", &phases[i], "float", ScopeIndex);
}
}

static
/* This routine simply lets the sampler run. An alternative, asynchronous
sampling strategy is to simply call ScopeCollectSignals() from this
routine
(at interrupt level). However, this scheme is probably more indicative of
the way most users will implement Scope. */
int ScopeDemoTimerInterrupt(void)
{
    semGive(SampleSemaphore);
    return(0);/* sysAuxClkConnect should take VOIDFUNCPTR */
}

static
/* Called by main loop to clean up mem usage */
void ScopeDemoShutdownInternal(void)
{
    if (tid != 0) {
        if (SampleSemaphore != NULL) {
            /* Must be aux clk, so clean up. */
            sysAuxClkDisable();
            sysAuxClkConnect(NULL, 0);
            semDelete(SampleSemaphore);
            SampleSemaphore = NULL;
        }
        if (ScopeNoServer == 0) {
            ScopeShutdown(ScopeIndex);
        } else {

```

```
        #if VXWORKS_AE_VERSION_1_0_OR_BETTER
            ScopeRemoveMultipleSignals("", ScopeIndex);
        #else
            ScopeRemoveMultipleSignals("*", ScopeIndex);
        #endif
        ScopeNoServer = 0;
    }

    ScopeIndex = -1;
    free(singroup);
    tid = 0;
}

/* Reset flag */
ScopeDemoShutdownRequest = 0;
}

static
/* This routine generates the signals, and does the sampling.
 * It attaches a semaphore to the Aux clock iff useAuxClock is non-zero.
 */
int ScopeDemoSampler(int requestedSR, int useAuxClock, int noCollection)
{
    int nTicksBwSamples = 0;
    float actualSR;

    if (useAuxClock) {
        SampleSemaphore = semBCreate(SEM_Q_PRIORITY, SEM_EMPTY);
    }

    /* Connect the timer to the auxillary clock interrupt. */
    if (useAuxClock &&
        (sysAuxClkConnect(ScopeDemoTimerInterrupt, 0) == OK)) {
        sysAuxClkEnable();
        sysAuxClkRateSet(requestedSR);
        /* hardware may not be exact */
        actualSR = sysAuxClkRateGet();
    } else {
        actualSR = sysClkRateGet();
    }
    if (actualSR < requestedSR) {
        /* Requested rate (requestedSR) is higher than
         * maximum achievable rate.
         */
        nTicksBwSamples = 1;
    } else {
        nTicksBwSamples = actualSR / requestedSR;
        actualSR = requestedSR;
    }
}

/* Tell scope the actual sample rate (hardware may not be exact). */
ScopeChangeSampleRateInt((int) actualSR, ScopeIndex);
```

```

        if((actualSR/requestedSR > 1.1) || (actualSR/requestedSR < 0.9)) {
            printf("ScopeDemo: requested rate (%f Hz) not \n"
                "achievable, actual sampling rate is %f Hz\n",
                (float) requestedSR, actualSR);
        }

        Dt = 1./actualSR;
        while(!ScopeDemoShutdownRequest) {
            if (useAuxClock) {
                semTake(SampleSemaphore, WAIT_FOREVER);
            } else {
                taskDelay(nTicksBwSamples);
            }
            Time += Dt;
            Sample(Time, noCollection);
        }
        ScopeDemoShutdownInternal();
        return(0);
    }

    /*
    ARGUMENTS
        If useAuxClock is TRUE, use the Aux clock. Otherwise (or if there is no
        Aux clock, use taskDelay.
        scopeIndex is the Data Monitor index to use. It must be coordinated with
        the GUI.
        verbosity controls the amount of warning and debug messages. 0 means
        only errors will be printed. Higher values causes more output.
        If reqDataBufSize is non-zero, use that value rather than the default
        (51k)
        If noServer is true, then this demo will not start a Scope
        server. Assumes
            one is already started.
        If noCollection is non-zero then this demo will not call
        ScopeCollectSignals
            since it is assumed that another process is collecting.
    */
    void ScopeDemo( int useAuxClock, int scopeIndex, int verbosity,
        int reqDataBufSize, int noServer, int noCollection )
    {
        int samplingRate = 60;    /* Hz */
        int dataBufSize, signalBufSize, eventBufSize;

        /* Initialize */

        if (ScopeIndex > 0) {
            return;
        }
    }

```

```
if (reqDataBufSize == 0) {
    dataBufSize = 17*8*384; /* 51k data buffer (ints/floats) */;
} else {
    dataBufSize = reqDataBufSize;
}
signalBufSize = 40960;
eventBufSize = -1;
if (noServer == 0) {

    /* Initialize an index. */
    ScopeIndex = ScopeInitServer(dataBufSize, signalBufSize, verbosity,
                                scopeIndex);
    if (ScopeIndex < 0) {
        printf("ScopeInitServer failed, return code = %d\n", ScopeIndex);
        return;
    }

    /* Attach an event buffer to that index. */
    ScopeHandle = ScopeEventsAttach(eventBufSize, ScopeIndex);
    if(ScopeHandle == 0) {
        printf("ScopeEventsAttach failed, exiting!\n");
        return;
    }

} else {
    ScopeIndex = scopeIndex;
    ScopeNoServer = 1;
}

ScopeIndex = scopeIndex;

InstallFakeSignals(8);

tid = taskSpawn("ScopeDemo", 100, VX_FP_TASK|VX_STDIO, 0x4000,
               (int (*)()) ScopeDemoSampler,
               samplingRate, useAuxClock, noCollection,0,0,0,0,0,0,0);
}

void ScopeDemoShutdown(void)
{
    if (tid != 0) {
        ScopeDemoShutdownRequest = 1;
    }
}
```

Makefile for vxdemo.c

Use this makefile as a template for compiling your code (or the **vxdemo.c** program) instrumented with ScopeProbe API.


```
#####
#
#           Data Monitor demonstration makefile.
#
# This makefile is provided as an example only! It compiles the
# VxWorks version 6.3 objects using the "ccpentium" gnu cross-
# compiler for solaris2. Users will of course have to modify
# this for their installations.
#
# We recommend you copy the current demo source directory before
# compiling. For instance:
#   mkdir mydir
#   cp -r ./src/scopedemo mydir/src
#   cd mydir/src
#   make
#
# The makefile will create a "mydir/lib" directory to hold your
# objects.
#
#####

# This makefile should work even if you have not properly installed your
# cross-compiler environment. However, the following variables must be set
# to reflect your configuration:

WIND_BASE = /local/VxWorks/VDT.3.0
WIND_HOST_TYPE = sun4-solaris2
ARCH = pentium
CPU = PENTIUM3

# The following script should automatically be able to compile Data Monitor
# demonstration files.
#
# Note 1: This makefile generates ".so" object files.
# Note 2: This makefile creates the USEROBJDIR directory, if it doesn't
# exist.

GCCHOME = $(WIND_BASE)/gnu/3.3.2-vxworks60/$(WIND_HOST_TYPE)
CC = $(GCCHOME)/bin/cc$(ARCH)

USEROBJDIR = lib/$(CPU)Vx6.3gcc3.3.2
SCOPE_INCLUDES = ../../include/share

# makefile rules

all:      $(USEROBJDIR) $(USEROBJDIR)/scopedemo.so

$(USEROBJDIR):
    mkdir -p $@
```

```
$(USEROBJDIR)/%.o: %.c
    $(CC) -Wall -O -c -DCPU=$(CPU) \
        -I$(WIND_BASE)/vxworks-6.3/target/h -I$(GCCHOME)/include \
        -I$(SCOPE_INCLUDES) -DRTI_VXWORKS $< -o $@

$(USEROBJDIR)/%.so: $(USEROBJDIR)/%.o
    $(GCCHOME)/bin/ld$(ARCH) -r $< -o $@
```

C.3 Source Code for Linux

The file, **scopedemo.c**, is an example of a simple application that uses Data Monitor. The example application plots a number of sinusoids and contains a simple control system with a simulated physical system that is noisy. It is meant only to illustrate some of the Data Monitor features.

The file and its makefile are located at:

```
WIND_SCOPETOOLS_BASE\target\src\scopedemo
```

where **WIND_BASE** (an environment variable of the same name) is the root of the tree where you installed the Run-Time Analysis Tools. The **scopedir\src\target\src\scopedemo** directory contains the **scopedemo.c** demo code file (see below), and a makefile (see [Makefile for vxdemo.c](#), p.332), to help you recompile the demonstration program.



NOTE: You must edit the **makefile** to have it reflect the file structure of your system.

Linux scopedemo.c Program

```
/* scopedemo \- UNIX and Windows NT scope demonstration program. */

/* Copyright Real-Time Innovations, Inc., 1989-91. All rights reserved.
   Permission to modify and use internally is granted to Data Monitor
   licensees,
   provided this notice is not removed. */
/*
   modification history
   -----
5.4a,08nov00,gah increased signalBufSize
5.3a,01dec99,laf Changed ThreadSleep to new API
5.1a,26oct98,nm ScopeInitServer() now takes two buffer sizes.
```

```
5.1a,16oct98,nm Updated to use the new thread creation api in rtilib.
5.1a,16oct98,nm usleep() is not needed.
5.1a,16jul98,laf usleep is defined on LINUX
5.1a,12mar98,nm Updated for scope 5.1 release.
5.0b,10Jul97,mlh Commented big time and it now build on Windows.
5.0b,24jun97,sas Added position & velocity gains to signals.
5.0b,19Jun97,mlh Now, scopedemo is uses threads for windows.
4.4a,17Mar97,mlh Added Ptr signal Install.
4.4a,29aug96,mlh Signal types changed.
4.3g,30jul96,mlh Added Different types.
                    Added TriggerImmediate , and TriggerDisabled.
RTI,09dec91,sas converted to mangel format.
RTI,06jul90,sas added simulator. Converted to ANSI-C
RTI,????89,sas Original code written by Stan Schneider in 1989.
*/

/*
USAGE:
    scopedemo samplerateinHz verbosity index

DESCRIPTION:
    "scopedemo" provides a simple Host-to-Host demonstration of Data
    Monitor.

    When executed, this program will present a simple menu of options to
    control the simulated control system, add and remove signals, etc.

    In summary, to run the UNIX-to-UNIX demo:
    on any workstation (say "targetHostname", running Solaris 2.5):
        cd scopedir/bin/sparcSol2.5
        scopedemo

    on any workstation (including "targetHostname"):
        scope[.x] targetHostname &

    This demonstration is very simple; it should, however, give
    you an idea of {\em Data Monitor's} capabilities.

COMPILING:
    This demonstration works on Windows 95/NT, and those flavors of UNIX
    that support threads.

NOTE:
    This code, and the libraries it uses, are available free-of-charge
    to Data Monitor licensees.

*/

/*
    This code runs as four threads:
    1) The user's simulation
    2) The menu parser
    3) Probe daemon
    4) Link daemon
```

```

    All user input (including the menu parser) is handled by the Query
    Package in this program (see "man Query").
*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/types.h>
#include <stddef.h>
#include <string.h>

#include "rtilib/rti_types.h"
#include "rtilib/query.h"
#include "rtilib/rti_osapi.h"
#include "scope/scope.h"

#define MAXPHASES(8)
#define PHASESHIFT(0.1)

#ifdef RTI_SUN4
    #define RAND_MAX      (32767)
#endif

/*
 * ThreadInfo_t provides information when threads are possible.
 */
typedef struct ThreadInfo {
    int index;
    int handle;
}ThreadInfo;

/*
 * Global variables. Using this many globals is not good coding style.
 * However, this code is intended to serve as a simple example...
 */

/*
 * As of Data Monitor 5.0, users can registers signals as offsets from a
 * pointer. TestStruct_t will be used to show the ScopeRegisterWithOffset
 * call.
 */
typedef struct TestStruct_t {
    float First;
    int Second;
} TestStruct_t;

TestStruct_t OffsetData[10] = {{0.0f, 0}, {0.1f, 0}, {0.2f, 0}, {0.3f, 0},
                                {0.4f, 0},
                                {0.5f, 0}, {0.6f, 1}, {0.7f, 1}, {0.8f, 1},
                                {0.9f, 1}};
```

```
/*
 * During this demo we will increment PtrTestStruct through the OffsetData
 * array, inorder to demonstrate offsets to signals.
 */
TestStruct_t   *PtrTestStruct = &OffsetData[0];

/*
 * These test the new scopelink which handles specific types.
 * Users can register signals with exact types without losing precision. If
 you
 * are building on an ALPHA make sure that RTI_ALPHA is set.
 */
double          Byte64Double          = 4.0;
float           Byte32Float            = 3.5f;
#ifdef RTI_ALPHA
int             Byte32Int              = 101;
unsigned int    Byte32UInt             = 0xffffffff;
#else
long           Byte32Int              = 101;
unsigned long   Byte32UInt             = 0xffffffff;
#endif
short          Byte16Int              = 2;
unsigned short  Byte16UInt            = 0xffff;
char           Byte8Int               = 'a';
float          FloatValue             = 2.0f;

/*
 * Support for pointers to signals was added in Data Monitor 5.0. We support
 any
 * number of dereferences.
 */
float          *PtrToFloatValue       = &FloatValue;
float          **PtrPtrToFloatValue   = &PtrToFloatValue;

double phases[MAXPHASES];
int NumberOfPhases = 0;

/*
 * A whole bunch to simulate a real system.
 */
double sint      = 0.0;
double sin2t     = 0.0;
double cost      = 1.0;
float Pos        = 0.0f;
float Vel        = 0.0f;
float Acc        = 0.0f;
float Posdes     = 1.0f;
float Veldes     = 0.0f;
float Kp         = 10.0f;
float Kv         = 4.0f;
float NoiseMag   = 0.1f;
float Time       = 0.0f;
float Dt         = 0.005f;

int StepPeriod = 300;
```

```
/*
 * SineWaves /- Calculate the sine values.
 */
void SineWaves( float t, int handle )
{
    register int j;

    sint = sin(t);
    cost = cos(t);
    sin2t = sin(2 * t);
    for(j=0; j<NumberOfPhases; j++) {
        phases[j] = sin(t + PHASESHIFT*j);
    }

    if((sint > 0.1) && (sint < 0.2)) {
        ScopeEventsCollect(handle, 1, "Sine-0.1-0.2", &sint, RTI_DOUBLE64ID);
    }

    if((cost > 0.5) && (cost < 0.6)) {
        ScopeEventsCollect(handle, 1, "Cosine-0.5-0.6", &cost, RTI_DOUBLE64ID);
    }
}

/*
 * Noise /- Return a uniform random variable between a and b.
 */
float Noise(float a, float b)
{
    float result;

    result = (float)(((float) rand() / RAND_MAX) - 0.5);
    return (float)(result * (b - a) + (a + b) / 2.);
}

/*
 * Plant /- This "plant" is an Euler double integrator.
 */
void Plant(float t)
{
    Vel += Acc * Dt;
    Vel += Noise(-NoiseMag, NoiseMag);
    Pos += Vel * Dt;
}

void Control( float t, int handle )
{
    static int stepCount = 300;
    if ((StepPeriod > 0) && (stepCount-- <= 0)) {
        stepCount = StepPeriod;
        Posdes = -Posdes;
        ScopeEventsCollect(handle, 1, "PosChangeEvent", &Posdes, RTI_FLOAT32ID);
    }
    Acc = Kp*(Posdes - Pos) + Kv*(Veldes - Vel);
}
```

```
/*
 * Offset /- Set the PtrTestStruct to the next array position.
 */
void Offset(float t)
{
    PtrTestStruct = &OffsetData[((int)(t/0.5))%10];
}

void InstallFakeSignals( int num, int index )
{
    register int i;
    char str[132];

    if(num > MAXPHASES) {
        num = MAXPHASES;
    }
    NumberOfPhases = num;

/*
 * ScopeRegisterSignal() specifies to the GUI the existence and
 * characteristics of a signal. ScopeActivateSignal() specifies that a
 * sample should be collected when ScopeCollectSignal() is called.
 * ScopeInstallSignal() is simply a convenience function that calls
 * ScopeRegisterSignal() then ScopeActivateSignal() signal. Take a look at
 * the html pages for ScopeRegisterSignal() and ScopeInstallSignal for a
 * complete list of types.
 *
 * Data Monitor supports hierarchial names. The signals with the prefix
 * "DifferentTypes/" will all exist in a folder called "DifferentTypes".
 */

    ScopeInstallSignal("DifferentTypes/8ByteDouble", "volts", &Byte64Double,
        "double", index);
    ScopeInstallSignal("DifferentTypes/4ByteFloat", "volts", &Byte32Float,
        "float", index);
    ScopeInstallSignal("DifferentTypes/4ByteInt", "volts", &Byte32Int, "int",
        index);
    ScopeInstallSignal("DifferentTypes/4ByteUInt", "volts", &Byte32UInt,
        "unsigned int", index);
    ScopeInstallSignal("DifferentTv index);
    ScopeInstallSignal("DifferentTypes/2ByteUInt", "volts", &Byte16UInt,
        "unsigned short", index);
    ScopeInstallSignal("DifferentTypes/1ByteChar", "volts", &Byte8Int,
"char",
        index);

/*
 * You should use offsetof macro inorder to find the offset of a
 * structure variable. If your OS does not support this macro then
 * take a look at the html pages for ScopeRegisterSignalWithOffset.
 * I explain a pretty fool proof way of calculating offset automatically.
 */
```

```
ScopeRegisterSignalWithOffset("Offset/First", "volts",
                              &PtrTestStruct, "float",
                              offsetof(TestStruct_t, First), index);
ScopeRegisterSignalWithOffset("Offset/Second", "volts",
                              &PtrTestStruct, "int",
                              offsetof(TestStruct_t, Second), index);
/*
 * We activate this signals immediately after the registration only
 * to show the new functionality. You can activate signals anytime
 * after you register the signal. Also, take a look at
 * ScopeDeactivateSignal().
 */

ScopeActivateSignal("Offset/First", index);
ScopeActivateSignal("Offset/Second", index);

/*
 * We also support pointers to any signal type. ScopeCollectSignals()
 * will dereference the pointer before sampling the signal.
 */

ScopeInstallSignal("Pointer/Float", "volts", &FloatValue, "float",
                  index);
ScopeInstallSignal("Pointer/FloatPtr", "volts", &PtrToFloatValue,
"float*",
                  index);
ScopeInstallSignal("Pointer/FloatPtrPtr", "volts", &PtrPtrToFloatValue,
"float**", index);

ScopeActivateSignal("Pointer/Float", index);
ScopeActivateSignal("Pointer/FloatPtr", index);
ScopeActivateSignal("Pointer/FloatPtrPtr", index);

/*
 * Here are all of the system variables.
 */
ScopeInstallSignal("Sine", "volts", &sint, "double", index);
ScopeInstallSignal("Cosine", "volts", &cost, "double", index);
ScopeInstallSignal("Sine2T", "volts", &sin2t, "double", index);
ScopeInstallSignal("Pos", "meters", &Pos, "float", index);
ScopeInstallSignal("PosDesired", "meters", &Posdes, "float", index);
ScopeInstallSignal("Vel", "m/s", &Vel, "float", index);
ScopeInstallSignal("Acc", "m/s/s", &Acc, "float", index);
ScopeInstallSignal("PosGain", "N/m", &Kp, "float", index);
ScopeInstallSignal("VelGain", "N/m/s", &Kv, "float", index);

for(i = 0; i < num; ++i) {
    sprintf(str, "SinGroup/sin(t+%.1f)", PHASESHIFT*i);
    ScopeInstallSignal(str, "volts", &phases[i], "double", index);
}
}
```



```
void SetRate(float rate, ThreadInfo *threadInfo)
{
    if (rate <= 0.f) {
        printf("Sample rate must be positive.\n");
        return;
    }

    ScopeChangeSampleRateInt((int) rate, threadInfo->index);
    Dt = (float) 1.0/rate;
    printf("Sample rate is %.2f Hz\n",rate);
}

void Reset(void)
{
    Pos = Vel = Acc = 0.0f;
}

void Menu( int index )
{
    printf("\n Scope index (%d)\n", index);
    printf(" ----- \n");
    printf(" h - Print menu\n");
    printf(" a - Add signals\n");
    printf(" r - Remove signals\n");
    printf(" v - Activate signals\n");
    printf(" w - Deactivate signals\n");
    printf(" d - Display registered signals\n");
    printf(" i - Display activated signals\n");
    printf(" p - Change position step size\n");
    printf(" l - Change length of step\n");
    printf(" n - Add noise\n");
    printf(" R - Change sampling rate\n");
    printf(" e - Throw event\n");
    printf(" q - Quit\n");
}

void ProcessCommand(char cmd, QueryInfo qi, ThreadInfo *threadInfo)
{
    static char removeStr[128] = "";
    static char eventStr[128] = "testEvent";
    static float sampleRate = -1.0f;

    if (sampleRate < 0) {
        sampleRate = (float) 1.0/Dt;
    }

    switch (cmd) {
        case 'h':
            Menu(threadInfo->index);
            break;

        case 'a':
            InstallFakeSignals(QueryInt("Number to install", 0, MAXPHASES, 1, qi),
                               threadInfo->index);
            break;
    }
```

```
        case 'r':
QueryString("Remove signals starting with", removeStr, qi);
ScopeRemoveMultipleSignals(removeStr, threadInfo->index);
break;

        case 'd':
ScopeShowSignals(threadInfo->index);
break;

        case 'i':
ScopeShowActiveSignals(threadInfo->index);
break;

        case 'p':
Posdes = (float)QueryAnyReal("Position Step Size", fabs(Posdes), qi);
break;

        case 'l':
StepPeriod = QueryInt("Step Length (0 => off)", 0, 1000000,
                      StepPeriod, qi);
break;

        case 'n':
NoiseMag = (float)QueryReal("Noise magnitude", 0.0, 10000.0, NoiseMag,
                             qi);
break;

        case 'R':
SetRate(sampleRate = (float)QueryReal("Sample rate", 3.0, 100.,
                                       sampleRate, qi), threadInfo);
break;

        case 'v':
QueryString("Activate signals starting with", removeStr, qi);
ScopeActivateMultipleSignals(removeStr, threadInfo->index);
break;

        case 'w':
QueryString("Deactivate signals starting with", removeStr, qi);
ScopeDeactivateMultipleSignals(removeStr, threadInfo->index);
break;

        case '\n':
break;

        case 'e':
QueryString("Event string", eventStr, qi);
ScopeEventsMessage(threadInfo->handle, 1, eventStr);
break;

        case 'z':
Reset();
break;
```

```
        case 'q':
            ScopeShutdown(threadInfo->index);
            break;

        default:
            printf("Bad command %c. Try 'h'.\n", cmd);
    }
}

void UsageError(void)
{
    printf("Usage: scopedemo samplerateinHz verbosity index\n");
    exit(1);
}

void *ScopeDemoSampler( void *param )
{
    ThreadInfo *threadInfo = (ThreadInfo *) param;
    float time = 0.0f;
    RTINtpTime rtiTime;

    for(time = 0.0f;; time += Dt) {
        SineWaves(time, threadInfo->handle);
        Plant(time);
        Control(time, threadInfo->handle);
        Offset(time);
        RtiNtpTimePackFromNanosec(rtiTime, (int) Dt,
            (int)((1000000000.0 * Dt) -
                (1000000000.0 * (double) rtiTime.sec)));
        RtiThreadSleep(&rtiTime);
        ScopeCollectSignals(threadInfo->index);
    }
    return NULL;
}

void QueryMenu( ThreadInfo *threadInfo )
{
    char cmd;
    QueryInfo qi;

    InstallFakeSignals(MAXPHASES, threadInfo->index);
    printf("\nData Monitor demonstration target simulator.\n");
    printf(" *****\n");
    printf("\nReal-Time Innovations, Inc.\n");
    Menu(threadInfo->index);

    qi = QueryCreate();
    QueryFlushChar(qi);
    cmd = '\0';
    while (cmd != 'q') {
        cmd = QueryAnyChar("Scopedemo", 'h', qi);
        ProcessCommand(cmd, qi, threadInfo);
    }
    QueryDestroy(qi);
}
```

```
int main(int argc, char *argv[])
{
    int index      = -1;
    float rate     = 30.f;
    int sampleBufferSize, eventBufferSize, signalBufferSize;
    int verbosity = 0;
    RTISystemThreadInfo threadSysInfo;
    RTIThreadOptions samplerThreadOptions = RTI_THREAD_PRIORITY_ENFORCE;
    ThreadInfo *threadInfo;
    RTIThreadId tid;
    /* This is normalized priority (range 0-255). We want to run the
     * sampling thread at a high priority.
     */
    int samplerThreadPriNorm = 200;
    int samplerThreadPriNative;

    if(argc >= 2) {
        if(sscanf(argv[1], "%f", &rate) != 1) {UsageError();}
    }
    if(argc >= 3) {
        if(sscanf(argv[2], "%d", &verbosity) != 1) {UsageError();}
    }
    if(argc >= 4) {
        if(sscanf(argv[3], "%d", &index) != 1) {UsageError();}
    }

    sampleBufferSize = 256 * 1024; /* 256k sample buffer
                                   (ints/floats/doubles) */
    eventBufferSize = -1;          /* Pick the default value */
    signalBufferSize = 64 * 1024; /* Approx. 200 signals, assuming name &
units                                for a signal is 23 characters (including
                                   two '\0' characters). */

    threadInfo = (ThreadInfo *) calloc(1, sizeof(ThreadInfo));

    threadInfo->index = ScopeInitServer(sampleBufferSize, signalBufferSize,
                                       verbosity, index);
    if(threadInfo->index < 0) {
        printf("ScopeInitServer failed, exiting!\n");
        exit(1);
    }

    threadInfo->handle = ScopeEventsAttach(eventBufferSize, threadInfo-
>index);
    if(threadInfo->handle == 0) {
        printf("ScopeEventsAttach failed, exiting!\n");
        exit(1);
    }

    printf("\nScope index (%d) successfully initialized!\n", threadInfo-
>index);
    SetRate(rate, threadInfo);
}
```

```
RtiSystemThreadInfoGet(&threadSysInfo);
if (threadSysInfo.realtimeEnabled) {
    samplerThreadOptions = (RTIThreadOptions)
        (samplerThreadOptions | RTI_THREAD_REALTIME_PRIORITY);
}
samplerThreadPriNative = RtiThreadNativePriorityGet(samplerThreadOptions,
    samplerThreadPriNorm);

tid = RtiThreadCreate("SamplerThread", samplerThreadPriNative,
    samplerThreadOptions, 16*1024, ScopeDemoSampler,
    threadInfo);

QueryMenu(threadInfo);
RtiThreadIdDelete(tid);
free(threadInfo);

return(0)
```

Makefile for scopedemo.c

Use this makefile as a template for compiling your code (or the **vxdemo.c** program) instrumented with ScopeProbe API.

```
# Unix makefile.public
# -----
# This makefile will compile the "scopedemo" demonstration program for UNIX.
#
# You will need an ANSI-compliant compiler (such as gcc) to compile the
# demonstration program.

# You must set 'ARCH' to the platform that you are building for, which
includes
# the OS and compiler versions. For example, set it to sparcSol2.8gcc2.95 for
# Solaris 2.8.x (gcc 2.95 compiler), i86Linux2.4gcc2.96 for RedHat 7.1-7.3
# (gcc 2.96 compiler), i86Linux2.4gcc3.2 for RedHat 8.0 (gcc 3.2 compiler),
# i86Linux2.4gcc3.2.2 for RedHat 9.0 (gcc 3.2.2 compiler) and
# pentium3MVCGE3.1gcc3.3 for MontaVista CGE3.1 (gcc 3.3 compiler).

ARCH = pentium3MVCGE3.1gcc3.3

CC = gcc

INCLUDELIST = \
    -I../include/share \
    -I../include/unix

LIBRARIES = ../arch/$(ARCH)/libscope711tcpz.a \
    ../arch/$(ARCH)/libxmlparsez.a \
    ../arch/$(ARCH)/libutilsipz.a

SYSTEMLIBS = -lm -lnsl -lpthread
```

```
# For Solaris platform, you will need following system libraries
ifneq (, $(findstring sparcSol, $(ARCH)))
SYSTEMLIBS += -lsocket -lposix4
endif

scopedemo: scopedemo.c
$(CC) $(CFLAGS) -DRTI_UNIX $(INCLUDELIST) scopedemo.c -o scopedemo $(LIBRARIES)
$(SYSTEMLIBS)
```

D

MATLAB and MATRIX_x

Examples

- [D.1 Introduction 347](#)
- [D.2 MATLAB Example 347](#)
- [D.3 MATRIX_x Example 351](#)

D.1 Introduction

This appendix presents example script files that can be used in **MATLAB** and **MATRIX_x** to plot signals saved by Data Monitor.

D.2 MATLAB Example

This section shows a **MATLAB** script file, **varplot.m**, that plots signals. It also shows a sample **MATLAB** session that loads a script file saved by Data Monitor, then uses the **varplot** script to produce a PostScript file.

Example D-1 Varplot m-File

```
% Name:
%   varplot    variable plot program for scoped matlab data
%
% Usage:
%   <load your scope data>
%   [define a global variable varplot_Names]
%   varplot
% Parameters:
%   varplot_Names: a matrix of signal names to plot.
%                   All rows must have the same number of characters.
%                   Example: ['ForceUp  ' ;
%                             'ForceDown']
%                   varplot_Names should be in order curves are plotted
%
%   subplotactive: The active subplot number (if any)
%                   if this variable exists, then handle subplot
%                   windows correctly
%
% Description:
%   If varplot_Names is defined, "varplot" will print the named
%   signals.
%   Otherwise, "varplot" will prompt you for signals to be
%   plotted, then plot
%   them, with labels, etc.
%
% See Also:
%   smartplot, plotall, legend
%
% Language:  PRO-MATLAB                                     Version 3.5
%
% Written by: Stan Schneider, Real-Time Innovations    July, 1988
%
% Revision History:
%
%_____

plotindex = [];
plotdata= [];
plotnames= [];
unitnames= [];
s = '  ';
if (exist('varplot_Names')),
    % find the number of curves
    Num_Curves=size(varplot_Names); Num_Curves=Num_Curves(1,1);
    this_Curve = 0;
end;

numsignals = length(data(1,:));

% If varplot_Names doesn't exist, prompt the user for signal names.
more = 1;
while(more == 1)
    if (~exist('varplot_Names')),
        s = input('Plot which signal? [end] ','s');
```



```

        if((length(s) == 3))
            if((s == 'end'))
                more = 0;
                break;
            end;
        end;
    else
        this_Curve = this_Curve + 1;
        if(Num_Curves == this_Curve),
            more = 0;
        end;
        s = varplot_Names(this_Curve,:);
    end;

    index = nameindex(names,s);
    if(index == 0)
        disp(s);
        disp(': Variable not found');
    else
        plotindex = [plotindex index];
    end
end;

% Build the arrays to be plotted.
for i=plotindex
    plotdata = [plotdata data(:,i)];
    plotnames = [plotnames ; names(i,:)];
    unitnames = [unitnames ; units(i,:)];
end
plotnames          % Display the plot names.
plot(time,plotdata);

xlabel('Time (seconds)');
ylabel(units(index,:));

if(exist('runtitle'))
    titlestring = [runtitle ' - ' timestamp];
else
    disp('warning - no runttitle');
    titlestring = [timestamp];
end
title(titlestring);

subnum = 0;
if(exist('subplotactive')),
    subnum = subplotactive;
end;

legend(plotnames, unitnames, 1, subnum, 1);
Example MATLAB Session

```

This **MATLAB** session loads the data saved as **dynamicPayload.mat** by running the **dynamicPayload.m** script, both created by Data Monitor when saving data in **MATLAB** format. The session then uses **varplot** to plot two variables. The

resulting plot is saved in encapsulated PostScript format. The figure produced by this example appears in the manual as:

```
>> dynamicPayload
```

```
notes =
```

```
Notes:
```

```
This is a slew with the uncontrolled dynamic payload.  
Both force sensor filters active, at 20Hz.  
No friction compensation active.
```

```
controller =
```

```
ComputedTorque
```

```
PDGains0 =
```

```
      4.0000      1.5000          0
```

```
PDGains1 =
```

```
      1.0000      0.5000          0
```

```
>> who
```

```
Your variables are:
```

DesForceX	ForceYUnfiltered	ans
DesForceY	PDGains0	controller
DesiredAccX	PDGains1	data
DesiredAccY	PosX	filename
DesiredPosX	PosY	names
DesiredPosY	RawForceProbeX	notes
DesiredVelX	RawForceProbeY	numberOfSamples
DesiredVely	SampleDivisor	numberOfSignals
ElbowPos	SampleRate	runtitle
ElbowTorque	ShoulderPos	time
ElbowVel	ShoulderTorque	timestamp
FILE_EXISTS	ShoulderVel	units
ForceX	TERM	
ForceXUnfiltered	VelX	
ForceY	Vely	

```
>> varplot
```

```
Plot which signal? [end] ForceX
```

```
Plot which signal? [end] ForceY
```

```
Plot which signal? [end] end
```

```
plotnames =
```

```
ForceX
ForceY

>> meta
>> !gpp metatmp -deps -fSaveMatlabFigure.ps
```

D.3 MATHWORKS Example

This section shows a **MATHWORKS** script file, **varplot.ms**, that plots signals. It also shows a sample **MATHWORKS** session that loads a script file saved by Data Monitor, then uses the **varplot** script to produce a PostScript file.

Example D-2 Varplot ms-File

```
# Name:
#   varplot.ms   variable plot program for scoped MATHWORKS data
#
# Usage:
#   <load your scope data>
#   [define a global variable varplot_Names]
#   execute file="varplot"
# Parameters:
#   varplot_Names: a matrix of signal names to plot.
#                   All rows must have the same number of characters.
#                   Example: ["ForceUp  ";
#                             "ForceDown"]
#                   varplot_Names should be in order curves are plotted
#
# Description:
#   If varplot_Names is defined, "varplot" will print the named
#   signals.
#   Otherwise, "varplot" will prompt you for signals to be
#   plotted, then plot
#   them, with labels, etc.
#
# See Also:
#   plot
#
# Language:  MathScript                                     Version 6.0
#
```

```
plotindex = [];  
plotdata = [];  
plotnames = [];  
unitnames = [];  
s = " ";  
If (exist(varplot_Names)),  
    # find the number of curves  
    Num_Curves=size(varplot_Names); Num_Curves=Num_Curves(1,1);  
    this_Curve = 0;  
endIf;  
  
numsignals = length(data(1,:));  
  
# If varplot_Names doesn't exist, prompt the user for signal names.  
more = 1;  
While(more == 1)  
    If (!exist(varplot_Names)),  
        s = getLine("Plot which signal? [end]");  
        If(length(s) == 3)  
            If(s == "end")  
                more = 0;  
                exit 0;  
            endIf;  
        endIf;  
    else  
        this_Curve = this_Curve + 1;  
        If(Num_Curves == this_Curve),  
            more = 0;  
        endIf;  
        s = varplot_Names(this_Curve,:);  
    endIf;  
  
    # find index of s within names  
    [rows, sz] = size(names);  
    sz = length(s); found = 0;  
    For i = 1:rows  
        If (sz == 0), exit 1; endIf;  
        If (stringex(names(i,1),1,sz) == s)  
            If (index(names(i,1)," ") == index(s," ") ...  
                | index(names(i,1)," ") == sz+1)  
                found = 1;  
                exit 1;  
            endIf;  
        endIf;  
    endFor  
  
    If (found == 0)  
        display(s + ": Variable not found");  
    else  
        plotindex = [plotindex, i];  
    endIf;  
endWhile;
```

```
# Build the arrays to be plotted.
For i=plotindex
    plotdata = [plotdata, data(:,i)];
    plotnames = [plotnames ; names(i,:)];
    unitnames = [unitnames ; units(i,:)];
endFor

display(plotnames)    # Display plot names

If (plotindex == [])
    return;
endIf;

If(exist(runtitle))
    titlestring = runttitle + " - " + timestamp;
else
    display("warning - no runttitle");
    titlestring = timestamp;
endIf

plot(time, plotdata, {title=titlestring, xlab="Time (seconds)",
    legend=plotnames + unitnames});
```

Example D-3 **MATRIX_x Session**

This **MATRIX_x** session loads the data saved as **dynamicPayload.xmd** by running the **dynamicPayload.ms** script, both created by Data Monitor when saving data in **MATRIX_x** format. The session then uses **varplot** to plot two variables. The resulting plot is saved in encapsulated PostScript format.



NOTE: In an actual **MATRIX_x** session, the input and output appear in different text areas. The listing below **merges** the two to give a sense of cause and effect, where the output of a command is shown following the command itself.

```
>> execute file="dynamicPayload"
```

```
notes (a column vector of strings) =
```

```
Session notes:
```

```
This is a slew with the uncontrolled dynamic payload.
```

```
Both force sensor filters active, at 20Hz.
```

```
No friction compensation active.
```

```
controller =
```

```
ComputedTorque
```

```
PDGains0 =
```

```
4.0000    1.5000    0
```

```
PDGains1 =  
  
    1.0000    0.5000    0  
  
>> who  
  
main:  
    data -- 1000x38  
    numberOfSamples -- 1x1  
    numberOfSignals -- 1x1  
    filename -- 1x1  
    runtitle -- 1x1  
    notes -- 2x1  
    buffernotes -- 1x1  
    SampleRate -- 1x1  
    SampleDivisor -- 1x1  
DifferentTypes_8ByteDouble -- 1000x1  
DifferentTypes_4ByteFloat -- 1000x1  
DifferentTypes_4ByteLong -- 1000x1  
DifferentTypes_4ByteULong -- 1000x1  
DifferentTypes_4ByteInt -- 1000x1  
DifferentTypes_4ByteUInt -- 1000x1  
DifferentTypes_2ByteInt -- 1000x1  
DifferentTypes_2ByteUInt -- 1000x1  
DifferentTypes_1ByteChar -- 1000x1  
    Pointer_Float -- 1000x1  
    Pointer_Float_ -- 1000x1  
    Pointer_Float__ -- 1000x1  
    Pointer_Int -- 1000x1  
    Pointer_Int_ -- 1000x1  
    Pointer_Int__ -- 1000x1  
    Pointer_Double -- 1000x1  
    Pointer_Double_ -- 1000x1  
    Pointer_Double__ -- 1000x1  
    Offset_First -- 1000x1  
    Offset_Second -- 1000x1  
    Pos -- 1000x1  
    PosDesired -- 1000x1  
    Vel -- 1000x1  
    Acc -- 1000x1  
    PosGain -- 1000x1  
    VelGain -- 1000x1  
    Sine -- 1000x1  
    Cosine -- 1000x1  
    Sine2T -- 1000x1  
    Square -- 1000x1  
SinGroup_sin_t_0_0_ -- 1000x1  
SinGroup_sin_t_0_1_ -- 1000x1  
SinGroup_sin_t_0_2_ -- 1000x1  
SinGroup_sin_t_0_3_ -- 1000x1  
SinGroup_sin_t_0_4_ -- 1000x1  
SinGroup_sin_t_0_5_ -- 1000x1  
SinGroup_sin_t_0_6_ -- 1000x1  
SinGroup_sin_t_0_7_ -- 1000x1  
    time -- 1x1000
```

```
names -- 38x1
units -- 38x1
timestamp -- 1x1
gains -- 1x2

>> execute file="varplot"

Plot which signal? [end] ForceX

Plot which signal? [end] ForceY

Plot which signal? [end] end

ForceX
ForceY

>> hardcopy file="SaveMatlabFigure.ps", {ps}
```


E

RKM Signal Definitions

E.1 Introduction	357
E.2 Signal Descriptions	357

E.1 Introduction

Remote kernel metrics (RKMs) are operating system signals (metrics) that are dynamically collected by the **RKM_monitor** (or **rkm_monitor_linux**) target agent. This appendix lists each available signal, arranged by the various types, and gives a complete description of the signal.

For information on the RKM feature, and using Data Monitor to display these signals, see [12. Displaying Remote Kernel Metrics](#)

E.2 Signal Descriptions

The nature of Data Monitor signals, as created and installed from your target code, is explained in [15. Installing Signals](#). The nature of RKM signals is not unlike the Data Monitor signals described there, except that RKM signals are values (metrics)

generated from the kernel environment in which the target program is running. These metrics can give you additional insight beyond just program variables, into the dynamics of interaction between your program and its environment.

The selection of metrics from the tables below, for inclusion on the command line, determines the output generated by the RKM monitor. The basic metric categories and the specific metrics you can choose in each category are listed in detail in [Table E-1](#) below. (Note: the "*" in the Metric column indicates these values are available by default.)

Table E-1 **Basic Categories of Metrics**

Category	Metric	Description
-sysmetrics [metrics]	* rtps	Number of RTPs in the system
	* tasks	Number of tasks in each state
	* memusage	Memory usage
	* objects	Number of WIND objects in system
	* cputime	CPU usage (requires "spyLib")
	network	Network stack data pools
	tcp	TCP statistics
	udp	UDP statistics
-rtpmetrics [metrics]	* tasks	Number of tasks in the RTP
	* memusage	Memory usage
-taskmetrics [metrics]	* stack	Stack usage
	* cputime	CPU usage (requires "spyLib")
-rtps [rtps]	<executable name>	Executable name, truncated to 63 characters
	<rtpid>[-<rtpid>]	Data Monitor Process ID or range of process IDs
-tasks [tasks]	<task name>	Task name, truncated to 63 characters

Table E-1 Basic Categories of Metrics (cont'd)

Category	Metric	Description
	<tid>[-<tid>]	Task ID or range of task IDs
	parent=<name>	Task owned by RTP <name>

Table E-2 lists the metric options you can include in the command line to modify the display characteristics of the RKM monitor as described.

Table E-2 Metrics Modifiers

Modifier	Description
-byrtp	Display metrics grouped by RTPs, and by tasks within an RTP
-bytask	Display metrics grouped by task
-bymetric	Display metrics grouped by metric
-percent	Display metrics as percents (instead of values)
-kbytes	Display metrics as kbytes (instead of bytes)
-total	Display metrics as cumulative totals
-spylib=[val]	Starts the "spylib" cpu utilization package (Spylib uses the Aux clock to sample "val" times per second)
-auxclock	Use the aux clock for timing
-samples=[val]	Sample Frequency: number of samples per second
-seconds=[val]	Sample Frequency: seconds between each sample
-microseconds=[val]	Sample Frequency: microseconds between each sample
-index=[val]	Data Monitor index (port) value (0-127)

E

Table E-2 **Metrics Modifiers** (cont'd)

Modifier	Description
-verbosity=[val]	Data Monitor verbosity level (0-3)
-samplebuf=[val]	Sample buffer size (in bytes)
-signalbuf=[val]	Signal description buffer size (in bytes)
-probe=[val]	Data Monitor scopeprobe daemon priority
-link=[val]	Data Monitor scopelink daemon priority
-monitor=[val]	RKM monitor task priority

Table E-3 shows the specific data items available for display in the Data Monitor GUI corresponding to the metrics options entered on the command line. The items for each metric appear as check boxes in the Signals Tree in the Data Monitor Signals Bar. All the signals for each metric type can be displayed by checking the node check box for that metric type, or each data item can be selected (or not selected) individually (see [Signals Bar](#), p.26).

Table E-3 **Data Displayed for Selected Metrics**

Metric	Data Item Displayed
-sysmetrics [rtps]	Total number of RTPs running
-sysmetrics [tasks]	" tasks " = total number of tasks running
	" ready " = total number of tasks in ready state
	" suspended " = total number of tasks in suspended state
	" stopped " = total number of tasks in stopped state
	" pending " = total number of tasks in pending state
	" delayed " = total number of tasks in delayed state
-sysmetrics [objects]	" sem_binary " = total number of waits for binary semaphores
	" sem_mutex " = total number of waits for mutex semaphores

Table E-3 Data Displayed for Selected Metrics (cont'd)

Metric	Data Item Displayed
	"sem_counting" = total number of waits for counting semaphores
	"sem_old" = total number of waits for old semaphores
	"sem_posix" = total number of waits for posix semaphores
	"queues" = number of accesses to system queues
	"queues_posix" = number of accesses to posix queues
	"RTPs" = total number of RTPs running
	"tasks" = total number of tasks running
	"watchdogs" = number of watchdogs encountered
	"file_handles" = number of file handles encountered
	"page_pools" = number of page pools encountered
	"page_managers" = number of page managers encountered
	"vmem_contexts" = number of vmem contexts encountered
	"timers_posix" = number of posix timers encountered
	"shared_data" = number of shared data accesses
-sysmetrics [cputime] (&& spy enabled)	"tasks" = amount of CPU time used by tasks
	"kernel" = amount of CPU time used by the kernel
	"interrupts" = amount of time spent in interrupt state
	"idle" = amount of time spent in idle state

E

Table E-3 **Data Displayed for Selected Metrics** (cont'd)

Metric	Data Item Displayed
-sysmetrics [memusage]	" RAM_allocated " = number of bytes of RAM allocated
	" RAM_free " = number of bytes of RAM deallocated
	" uVM_allocated " = number of bytes of micro-virtual memory allocated
	" uVM_free " = number of bytes of micro-virtual memory deallocated
	" kVM_reserved " = number of kbytes of virtual memory reserved
-sysmetrics [network]	" data_buffers " = number of data buffer accesses operations
	" data_free " = number of data buffer free operations
	" data_drops " = number of times data has been dropped
	" data_waits " = number of times access to data has had to wait
	" data_drains " = number of times data has been drained
	" system_buffers " = number of system buffer accesses
	" system_free " = number of system buffer free operations
	" system_drops " =
	" system_waits " = number of times operations have had to wait for system resources
	" system_drains " = number of times the system has been drained
-sysmetrics [sys tcp] (&& tcp_head)	" send_buffers " = number of bytes transmitted by TCP send buffers

Table E-3 **Data Displayed for Selected Metrics** (cont'd)

Metric	Data Item Displayed
	" recv_buffers " = number of bytes received by TCP receive buffers
-sysmetrics [sysupd] (&& upd_head)	" send_buffers " = number of bytes transmitted by UPD send buffers
	" recv_buffers " = number of bytes received by UPD receive buffers

Examples

The following examples may be helpful in understanding how the elements of the two tables above can be combined into command-line arguments input by the RKM monitor utility.

- Monitor the default system, RTP, and task metrics, 1 sample every 10 seconds:

```
->RKM_monitor -seconds-10
```

- Monitor just system, CPU usage, and task states, 1 sample per second:

```
->RKM_monitor -index-126 -samples-1 -system cputime tasks
```

- Monitor stack usage for every task created by the "testbed.vxe" RTP. 10 samples per second:

```
->RKM_monitor -index-125 -samples-10 -tasks parent-root -metrics  
stack value=0=0x0
```

- Start an RKM monitor with index 113 to monitor only the system metric tracking the number of tasks on the system, taking 10 samples every second:

```
->RKM_monitor -index=113 -sysmetrics tasks -samples=10
```

- The following sequence of commands starts an RKM monitor with a non-default port, shows that it is running, then stops it:

```
->RKM_list  
value = 0 = 0x0  
->RKM_monitor -index=120  
value = 0 = 0x0  
->RKM_list  
Monitor[120] is running  
value = 0 = 0x0  
->RKM_stop 125  
value = 0 = 0x0  
->RKM_list
```

value = 0 = 0x0

F

Glossary

This Glossary contains definitions for some of the common terms used throughout this manual.

active signals

These are **registered** signals that are set up on the host by the Signal Manager using the API call **ScopeActivateSignal()**. They appear in the **Signals Bar** of each data-display window. For further context, see [Signals Definitions](#), p.19.

annotation

Text (any comment you want) entered at a specified point on a **Plot** or **Plot XY** window grid area, using a popup menu. For detailed information, see [Annotations](#), p.77.

asynchronous sampling

Takes a snapshot of the variable values at regular intervals during program execution. Asynchronous sampling is often desirable because it is easy to set up and requires no changes to your application code. For detailed information, see [Sampling Signals](#), p.255.

Auto Fit

A toolbar button that, when selected, causes a **Plot** or **Plot XY** window to automatically zoom to fit when a signal goes off the screen. See [3.2 Toolbars](#), p.40.

buffer time

The number of seconds of data to show in the plot of a **Plot** or **Plot XY** window before refreshing. For detailed information, see [1.General View](#), p.55.

comm plugin

The plug-ins you use for communications between the host GUI and the target. For further context, see [3.Comm Plug-ins View](#), p.58.

data-display window

The four primary Data Monitor windows that display collected data. They include the **Plot**, **Plot XY**, **Dump Plot**, and **Monitor** windows (Chapters [7](#), [8](#), [9](#), and [10](#) respectively).

Data Monitor API

The real-time data-collection and signal-management module of Wind River Data Monitor that runs on the target platform. It collects the time history of variables in your real-time program, and is described in detail in [16. API Introduction](#).

derived signal

A new signal you create whose value is computed by mathematical operations on other, existing signals. For detailed information, see [3.3.10 Derived Signals](#), p.52.

display accuracy

Controls the accuracy of the grid line markers by setting the number of places to the right of the decimal point in the markers, in **Plot** and **Plot XY** windows. For detailed information, see [Properties Tab View](#), p.118 and [Default Plot Properties Panel](#), p.135, as well as the corresponding topics in each of the other data-display windows.

downloadable kernel module (DKM)

Stored VxWorks projects used to manage and build modules that you want to exist in the kernel space. For further context, see [On a VxWorks Target](#), p.200.

ellipsis ("...")

On all menu items, an ellipsis indicates that the option opens another window that requires further response or interaction before any action takes place.

event

An event is a specific type of signal. It appears under the **Event** tree branch in the **Signals Bar** for a target; this branch contains a list of all the events that were thrown by the target program. The values collected when a certain event is thrown are treated like a sample and the samples are joined by lines to make signals. For further context, see [7.5 Displaying Events](#), p.130.

Events API

A set of low-overhead logging routines that can be useful to monitor real-time systems with minimal effect on the timing behavior. For detailed information, see [16.5 Data Monitor Events API](#), p.260.

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 Data Monitor is running, which receives and processes the allocation record data collected from the target agent machine.

history limit

A parameter that sets how many lines of historical data to maintain and display in a **Dump** or **Mini-Dump** window.

index

Distinguishes the different instances of ScopeProbe daemons running on the same target. Up to **128** different instances may be started on a target, so the index can range from **0** to **127**. For more information, see [Scope Index](#), p.249.

installed signal

Data Monitor signals that are registered and activated (see [Signals Definitions](#), p.19). For further context, see [Usage Notes](#), p.18.

live signal

Live signals are those that are being displayed in a data-display window as they are being collected and analyzed. This is in contrast to signals displayed from a

snapshot, which are now static and can be saved in a file. For further context, see [The Snapshot Process](#), p.186.

marker

Shows the coordinates of a point on a plotting grid using a preset, or otherwise predetermined graphic symbol. For more information, see [3.6 Screen Operations](#), p.76 and [Events Collected as Markers](#), p.131.

measurement

A line on the plotting grid that measures the distance between any two points. For more information, see [On-grid Measurements](#), p.78.

Mini-Dump window

A scaled down version of the **Dump Plot** window (described in [9. The Dump Plot Window](#)), listing the value of each signal at each sampling, letting you see a running history of selected signal values scrolling through the window with time. For more information, see [Mini-Dump Window](#), p.28.

Mini-Monitor window

A scaled down version of the **Monitor** window (described in [10. The Monitor Window](#)), letting you see the current value of, and modify, target data in a static but dynamically updated list format. For more information, see [Mini-Monitor Window](#), p.28.

preference

Values you can select to change the physical window appearance properties for the current data-display window being viewed, or to change properties for the individual signal itself (as contrasted with **preferences**, described above). These properties are described in each of the respective data-display windows where they are configured. For example **Plot** window appearance properties, see [Properties Tab View](#), p.118, and for signal properties, see [7.3 Signal Properties Dialog Box](#), p.123, or the corresponding topics in **Plot XY** window (for **Dump Plot** and **Monitor** windows, only the **Properties** tab view is available, as there is no plotted signal). **Properties** values apply only to the window in which they are set.

process

Beyond the common usage meaning a sequence of steps to accomplish something, the term is also used to mean the same thing for a Linux program as **task** means

for a VxWorks program, that is, a running program, or a subdivision of a running program.

properties

Values you can select in a dialog box or tab view that change the physical appearance properties for a data-display window (as contrasted with **preference**, described above). These properties are described in each of the respective data-display windows where they are configured, for example, [7.6 Setting New Plot Window Preferences](#), p.134 for a **Plot** window, or the corresponding topic in any of the other three data-display windows.

registered signal

A signal on which you have used the API call **ScopeRegisterSignal()** to let Data Monitor know it exists (the first step in creating an active signal - see [Signals Definitions](#), p.19). For further context, see [Usage Notes](#), p.18.

remote kernel metrics (RKM)

Operating system signals (metrics) that are dynamically collected by the **RKM_monitor** (or **rkm_monitor_linux**) target agent, and displayed in real-time in the Data Monitor GUI. For further context, see [12. Displaying Remote Kernel Metrics](#).

RKM monitor

A program you must build in VxWorks, using the tools and procedures described in [12. Displaying Remote Kernel Metrics](#), and containing directives to display some specific operating system metrics of your choice when run from the Data Monitor GUI.

routine

A self-contained code module that can accept input, execute, and produce output; used interchangeably with function.

RTP (VxWorks Only)

A **VxWorks** real-time process, running in a protected environment.

scope index

An integer value, ranging from **0** to **127**, supplied by each target and used as an internal connection identifier by Data Monitor, allowing simultaneous connection

with multiple (up to 128) targets. This parameter is described in [2.3 Starting Data Monitor](#), p.10.

signal

A variable from your target program as installed and viewable in the Data Monitor GUI (see [Usage Notes](#), p.18). In Data Monitor, you give signals names by which you want to track them in the GUI.

signal installation

The process of naming the variables in your program you want to view, and preparing for viewing in the Data Monitor GUI. For more information, see [Installing Signals](#), p.84.

Signal Manager

The Data Monitor utility that manages the installation of signals from your target program that you want to view in the Data Monitor GUI. The process is described in detail in [4. Using the Signal Manager](#).

signal pair

Signal selection for **Plot XY** windows differs from signal selection for the other types of data-display windows, because each plotted line requires two signals. These signal pairs are created using the **XY Signals** dialog box, as described in [8.2 Creating XY Signal Pairs](#), p.140.

Signals Bar

A sub-window in each of the four data-display windows, with tab views in which you can select signals for viewing, and modify certain parameters. They are described in detail in each data-display window chapter.

snap

The action in which your drawing line jumps precisely to the data line in a **Plot** (only) window when you release the mouse button after dragging it, during a measurement operation. The option, and the maximum snap distance can be selected as described in [7.6 Setting New Plot Window Preferences](#), p.134.

snapshot

An selected operation that saves all the collected and analyzed data for all active signals, and for all connected targets, since the start of data collection in this Data

Monitor session. You can display snapshots in the **Plot** and **Plot XY** windows, along with live data and other snapshots. They are described in detail in [11. Working with Snapshots](#).

strip chart

This option changes the behavior of the graph area of a **Plot** window so that it presents a continuous scrolling plot, instead of repainting the plot every 20 seconds. This feature is described in detail in [Strip Chart](#), p.122.

synchronous sampling

A task run on the target, in response to selecting **Spawn sampler task** in the **Data Monitor Setup Options** dialog box, that collects data for any signals that you install. This task repeatedly calls **ScopeCollectSignals()**, based on its own timing, so it is asynchronous with the running of your application. For further context, see [Automatic Loading and Running](#), p.213.

timestamp

The date/time recorded on the collection of a signal or event. Timestamps are discussed in [7.5 Displaying Events](#), p.130, [8.4 Signal Properties Dialog Box](#), p.154, [9.2 Dump Plot Window Tour](#), p.164, [11.2 Utilizing Snapshots](#), p.185, and [Signals vs. Events](#), p.262

trigger

A facility in Data Monitor that provides control over when and how often samples are collected. Collection and plotting of samples begins when the trigger's **Start** condition is met (that is when the trigger fires), and continues to be plotted on the GUI until the **Stop** condition occurs. For further context, see [5. Triggering](#).

Trace Log

Events are recorded in a log file, which you can display in the **Trace Log** window, as well as save to disk. This file is described in [3.3.11 Trace Log Window](#), p.53.

verbosity

Controls the type and number of messages generated by the target server connection process. Using the default value of **0** generates only error messages. Specifying a larger value (in the range of 1-3) generates an increasingly greater variety and volume of messages. For specific information on setting verbosity value, see [VxWorks Data Monitor Setup Options Dialog Box](#), p.12.

Index

A

- accuracy
 - Dump Plot windows 172
 - Monitor Window 183
 - Plot Windows 119, 135, 153, 161, 170, 180
 - Plot windows 121
- active signals
 - creating 82
 - defined 19, 232
 - examples of activating 251
 - installing 250
 - setting up 49, 81
- adding markers 76
- agent, collection, architectural description 2
- annotations
 - adding in Plot Windows 77
- API, how to use 267, 295
- architecture 2
- ASCII snapshots 191
- aspect lock, in Plot XY window 148, 154
- asynchronous sampling
 - described 255
 - example 256
 - task 214
- auxiliary clock 215
- Axis Properties dialog box 43

B

- browse button 366
- browse button, snapshot dialog 192
- buffers, snapshots 258
- building
 - header files 212, 226
 - include files 212, 226

C

- calculating offsets 252
- capabilities 1, 9
- changing base file name 192
- collecting signals 255
- collection agent, architectural description 2
- colors
 - alt method to change trace lines 125, 156
 - dialog box 124
 - in Legend tab view 117, 150
 - in Legend window 27
 - in plot windows 151
 - in Signal Properties dialog 124, 156
 - in Signals tab view 116, 150
 - on left axis (Has Ruler option) 127, 159
 - preferences 56
 - selecting trace line colors 110, 144
 - signal trace lines 107

- table coordinated with Signals Tree 169
- used in snapshots 136, 162
- communications plug-ins, preferences 58
- configuration files
 - loading from disk 46
 - saving to disk 46
- configuring Data Monitor 214
- connection
 - problems 221
 - status 33
 - to targets 45
- context sensitive menu 71
- creating
 - derived signals 98
 - XY signal pairs 140
- cycle numbering 192

D

- daemons
 - manually starting 218
 - roll of, in buffer overflow 258
 - task descriptions 5
- data
 - collection, status of 87
 - display windows, introduction 19
- Data Monitor
 - API reference section 267, 295
 - architecture 2
 - exiting 69
 - features 1, 9
 - host-side GUI 81, 85
 - icons on Workbench 11
 - main windows 19
 - reference information 267, 295
 - running 10
 - Setup Options dialog box 45
 - starting automatically 11
 - starting manually 14
 - target application 5
 - target communication 5
 - VxWorks API description 3
- deactivating signals 254
- deleting markers 76
- demo program, starting with ScopeDemo 32
- demonstration target program, running 30
- derived signals
 - creating 98
 - dialog box 41, 52, 98
 - mathematical operators 104
 - troubleshooting 105
 - wizard 52, 98–104
- DFW
 - in Linux architecture 3, 4
 - in VxWorks architecture 2
- dialog box
 - Axis Properties 43
 - Colors 124
 - Data Monitor Setup Options 45
 - Derived Signals 52
 - derived Signals 41
 - New Target Connection 45
 - Open 46, 60, 61
 - Preferences 41, 53, 110, 134, 144
 - Print 112, 146
 - Save Config 46
 - Signal Properties 73, 110, 123, 144, 150, 152, 154
 - Triggering 50
 - XY Signals 41, 51, 143
- disconnecting targets 83
- display accuracy
 - Dump Plot windows 172
 - Monitor Window 183
 - Plot Windows 119, 135, 153, 161, 170, 180
 - Plot windows 121
- displaying events 130
- docking toolbars 40
- downsampling 88, 92, 258
- Dump Plot window
 - description 24, 164
 - display accuracy 172
 - history limit 172
 - preferences 61
 - resolution 172
 - table size 172

E

- ellipsis button 366
- environment variables
 - PATH 18
- error log 17
- events
 - displayed as markers 132
 - displayed as messages 133
 - displayed in the Signals Bar 130, 367
 - displaying 130
 - ScopeEventsCollect 130, 261
 - ScopeEventsMessage 130, 261
 - vs. signals 262
- Events API
 - described 260
 - in the demonstration program 30
 - setting up 260
 - using 261
- examples
 - activation 251
 - asynchronous signal sampling 256
 - events displaying 131
 - installing signals 253
 - MATLAB 347–351
 - MATRIXx 351–355
 - registration 251
 - registration with offset 253
 - ScopeCollectSignals() 256, 257
 - scopedemo.c 334–346
 - ScopeInstallSignal() 253
 - signal installation 36
 - synchronous signal sampling 257
 - vxdemo.c 324–334
 - Workbench target script 220
- exiting Data Monitor 69

F

- features 1, 6, 9
 - overview 53
 - preferences 53
 - triggering, overview 50
- File menu item 44

- file name extensions 192
- functions
 - see also* routines

G

- graph coordinates 76
- graphical user interface (GUI)
 - defined 367
 - host-side 81, 85
- grid line spacing 120, 135, 153, 161, 171

H

- header files 212, 226
- hierarchical naming of signals 251
- history limit 172
- host
 - defined 367
- host-side GUI 81, 85

I

- include files 212, 226
- initialization
 - Data Monitor 18
- installed signals
 - defined 19, 233
 - using all by default 49
- installing signals
 - Data Monitor API 249
 - examples 253
 - first 82
 - from command shell 36
 - in one step 250
 - organizing signal names 243
 - Signal Manager window 84
- instrumenting
 - alternative to 241
 - code 241
- introduction 1, 9

K

keep window on top 70

L

Legend tab view 117, 150

libraries

libscope.so 218

libscopewtxonly.so 218

libutilsip.so 218

libutilsnoip.so 218

vxdemo.so 218

Link Daemon

buffer overflows 258

task description 5

live buffer 259

loading

automatic 213

configuration files 46

manual 217

snapshots 193

log

file 53

window

see Trace Log

M

markers 76

MATLAB

example 347–351

snapshots 191

MATRIXx

example 351–355

snapshots 191

maximum snap distance 121, 136, 154

measurements 121, 122, 136, 154, 181

menu

descriptions 43

File 44

Plot 69

View 70

Window 70

Mini-Dump window

default properties 136

description 28

Mini-Monitor window

default properties 137

description 28

minimum grid line spacing 120, 135, 153, 161, 171

modifying signals 181

Monitor Window

description 25, 174

display accuracy 183

preferences 65, 182

resolution 183

writing data to target 181

multiple target connections

feature, described 7

unpredictable results with 209

with RKMs 203, 205, 208, 209

N

naming of signals 251

New Target Connection dialog box 45

O

offsets, calculating 252

on-grid menu 71

online documentation reference signals 254

Open dialog box 46, 60, 61

opening snapshots 193

options, triggering 92

overflow prevention 258

P

panning in Plot Windows 77

PATH environment variable 18

Pause 167, 177, 178

- Plot menu 69
 - plot plug-ins, preferences 60
 - Plot Window
 - adding annotations 77
 - adding markers 76
 - description 20, 108
 - display accuracy 119, 135, 153, 161, 170, 180
 - displaying signal values 109
 - Legend tab view 117, 150
 - max snap distance 121, 136, 154
 - min grid line spacing 120, 135, 153, 161, 171
 - panning 77
 - pop-up menu 71
 - preferences 63, 134, 171, 182
 - previous zoom 71, 115, 148
 - resolution 120, 121, 135, 153, 154, 161, 170
 - selecting a signal 110
 - snap measure to signals 121, 136, 154
 - snapshots 136, 162
 - status bar 78
 - taking on-grid measurements 78
 - toolbar 41
 - X offset 135, 161
 - X range 135, 161
 - Y offset 119, 135, 153, 161
 - Y range 135, 161
 - Y scale 119, 153
 - zooming 22, 76
 - Plot window
 - display accuracy 121
 - resolution 121
 - snap measure to signals 122, 181
 - strip chart 122
 - Plot XY Window
 - colors 151
 - description 22, 141
 - preferences 67, 160
 - previous zoom 71, 115, 148
 - pop-up menus
 - deleting snapshots 196
 - description, Legend tab view 74
 - description, on-grid 71
 - description, on-trace 73
 - description, Signals Tree 73
 - Plot window, Legend tab view 118, 151
 - Plot window, Signal Properties dialog box 124
 - Plot window, Signals tab view 117
 - Plot XY window, Signal Properties dialog 154
 - Plot XY window, Signals tab view 150
 - to disconnect GUI from the target 84
 - preferences
 - colors 56
 - comm plug-ins 58
 - dump plot window 61
 - feature overview 53
 - Monitor Window 182
 - monitor window 65
 - plot plug-ins 60
 - Plot Window 134, 171, 182
 - plot window 63
 - Plot XY Window 67, 160
 - scope.ini file 263
 - Preferences dialog box 41, 53, 110, 134, 144
 - preventing overflows 258
 - previous zoom
 - in Plot Windows 71, 115, 148
 - in Plot XY Windows 71, 115, 148
 - Print dialog box 112, 146
 - Probe Daemon, task description 5
 - processsampler 241
 - Properties tab 27
- ## R
- Real-time Process
 - see* RTP
 - registering signals 250
 - registration
 - examples 251
 - examples with offset 253
 - remote kernel metrics
 - building a monitor program for Linux 203
 - building a monitor program for VxWorks 200
 - introduction 199
 - port number 203
 - running multiple RKM monitors 203
 - troubleshooting 208
 - unpredictable results 209
 - using 206

- viewing on a Linux target 207
- viewing on a VxWorks target 206
- removing
 - markers 76
 - multiple signals 254
 - signals 254
- resolution
 - Dump Plot window 172
 - Monitor Windows 183
 - Plot Windows 120, 121, 135, 153, 154, 161, 170
 - Plot windows 121
- RKM
 - see* remote kernel metrics
- routines
 - Data Monitor API library overview 248
 - defined 369
 - triggering 259
- RTP
 - architecture including RTPs 2, 3
 - connection type, for instrumenting 13, 18
 - defined 369
 - installing an RTP signal 234
 - removing an installed RTP signal 242
- running
 - Data Monitor 10
 - demonstration target program 30
 - with a Linux target 225
 - with a VxWorks target 211

S

- sampling
 - asynchronous signals 255
 - downsampling 88, 92, 258
 - functions 259
 - rate 255
 - ScopeChangeSampleRate() 255
 - signals 255
 - synchronous signals 257
- Save Config dialog box 46
- save.ssc, workspace state file 17
- saving
 - configuration files 46
 - snapshots 45

- scope index
 - and event buffers 249
 - described 249
 - example code 251
 - Legend tab view 118, 151
 - limits 250
 - Signal Manager window 84
 - triggering example 94
- scope.ini file 263
- ScopeActivateMultipleSignal() 255
- ScopeActivateSignal() 250, 252, 255
- ScopeChangeSampleRate() 88, 92, 255, 258
- ScopeCollectSignals() 88, 92, 222, 223, 251, 254, 255, 256, 257, 258, 259
- ScopeDeactivateMultipleSignals() 255
- ScopeDeactivateSignal() 254, 255
- ScopeDemo function, starting with 32
- scopedemo.c example 334–346
- scopeIndex 268, 296
- ScopeInitServer() 222
- ScopeInstallArray() 255
- ScopeInstallSignal() 250, 255
- ScopeInstallSignalWithOffset() 250, 253, 255
- ScopePrintVersion() 222
- ScopeRegisterArray() 255
- ScopeRegisterSignal() 250, 251, 254
- ScopeRegisterSignalWithOffset() 250, 252, 254
- ScopeRemoveMultipleSignals() 254, 255
- ScopeRemoveSignal() 254, 255
- ScopeSamplerTaskCreate() 219
- ScopeShowActiveSignals() 255
- ScopeShowSignals() 255
- screen operations 144
- selected signals
 - defined 19, 233
 - in Dump Plot window 24, 165
 - in Legend Bar 27, 110, 144
 - in Monitor window 25, 174
 - in Plot window 20
 - in Plot XY window 22
 - in Signal Manager 83
 - in Snapshot window 112, 136, 146, 162, 166, 176, 186
 - in state info of data-display windows 48
 - verifying target connection 217

- setting up
 - active signals 49, 81
 - APIs 260
- setup options 214
- show snapshot 167
- signal installation
 - organizing signal names 243
 - Signal Manager window 84
- Signal Manager 49, 81
- Signal Properties dialog box 73, 110, 123, 144, 150, 152, 154
- signals
 - activating 82
 - activation 49, 81, 250, 252, 255
 - activation examples 251
 - collection 88, 92, 254, 255, 258
 - deactivate 255
 - deactivation 254
 - downsampling 88, 92, 258
 - install array 255
 - installation 82, 249, 255
 - installation with offset 253, 255
 - installing 36, 249
 - multiple activation 255
 - multiple deactivation 255
 - naming 251
 - online documentation reference 254
 - pairs 140
 - register array 255
 - registration 250, 251, 254
 - registration with offset 250, 252, 254
 - removal 255
 - remove multiple signals 255
 - removing 254
 - removing multiple 254
 - sampling 255
 - show active 255
 - show signals 255
 - vs. events 262
 - writing to target 181
- Signals Bar
 - description 26
 - events displayed in 367
 - menu item 114, 148, 167, 177
 - using 82
- Signals tab 27
- signals trees
 - described 27
 - using 82
- slope trigger parameter 88, 89
- snap distance 121, 136, 154
- snapping measures to signals 121, 122, 136, 154, 181
- snapshots
 - ASCII formatting 191
 - automating 189
 - behavior 136, 162
 - buffering 258
 - building file names 193
 - changing base file name 192
 - colors used 136, 162
 - cycle numbering 192
 - description of process 186
 - file name extensions 45, 192
 - format 191
 - loading 45, 193
 - MATLAB formatting 191
 - MATRIXx formatting 191
 - preferences 136, 162
 - saving to disk 45, 187, 191
 - taking 136, 162, 187
 - time stamping 192
 - triggering 93
- starting
 - demo program, type "ScopeDemo" 32
 - trigger 91
- starting the Data Monitor GUI
 - automatically 11
 - manually 14
- starvation 221, 223
- status bar
 - control of viewing 113, 147, 167, 177
 - Plot Windows 78
 - used to open Log window 53
- stop
 - trigger 91
- strip chart mode, Plot window 112, 122
- synchronous sampling 257

T

tabs

- Properties 27
- Signals 27

taking on-grid measurements

- demonstration program 36
- in Plot Windows 78

target

- application 5
- code, instrumenting 241
- connecting to 45
- disconnecting 83
- Linux 225
- names 16
- processsampler 241
- programming language note 5
- shell script 220
- writing data to 181

terminology 18

time stamps

- feature of data storage 6
- in definition of "Timing" 263
- in Dump Plot window 24, 164
- in MATLAB example 350
- in MATRIXx example 355
- in Snapshot window 192, 195
- in Varplot-m File example 349

tLinkDaemon task

- connection failure 221
- connection, but no data 222
- verifying target initialization 217

toolbar

- button descriptions 40, 41
- captions 115, 148
- docking 40
- Main 114, 147, 167, 177
- Plot Window 41, 113, 168, 178
- Plots 114, 148, 168, 178

toolbars

- Main 115, 148
- Strip Chart 115

ToolTips 40

tProbeDaemon task

- connection failure 221

- verifying target initialization 217

trace log 53

Trace Log window 53

trigger

- configuring 85
- feature, full description 85
- functions 259
- overview 50
- parameters 87, 88
- rearming 89, 94, 95
- snapshot 89, 94
- status 87
- tips 258

triggering

- options 92

Triggering dialog box 50

triggers

- rearming 95
- snapshot 92, 95
- start parameters 91
- status 91
- stop parameters 91

troubleshooting

- connection failures 221
- derived signals 105
- empty signals tree 83
- loading errors 220
- multiple target connections 209
- no data appears 222, 259
- no response from target 221
- overflows 258

U

using

- Events APIs 261
- installed signals by default 49
- log file 53
- Signals Bar 82
- signals trees 82

V

verbosity [16](#)
 defined [371](#)
 Trace Log window, effect on [53](#)
 Warning [13](#)
 View menu [70](#)
 vxdemo.c example [324–334](#)
 VxWorks targets [3](#)

W

WARNING
 target verbosity [13](#)
 Window menu [70](#)
 wizard, derived signals [52](#)
 Workbench icons [11](#)
 Workbench targets
 automatic loading and running [213](#)
 example script [220](#)
 manual loading and running [217](#)
 setup options [214](#)
 writeback
 feature described [181](#)
 to write data to target [181](#)
 writing to target [181](#)
 WTX
 protocol [5, 17](#)
 target server [17](#)

X

X
 offset [135, 161](#)
 range [135, 161](#)
 XY Signals
 creating XY signal pairs [140](#)
 dialog box [41, 51, 143](#)

Y

Y
 offset [119, 135, 153, 161](#)
 range [135, 161](#)
 scale [119, 153](#)

Z

zoom to fit
 in Plot Windows [114](#)
 in Plot XY Windows [147](#)
 zooming in Plot Windows [22, 76](#)